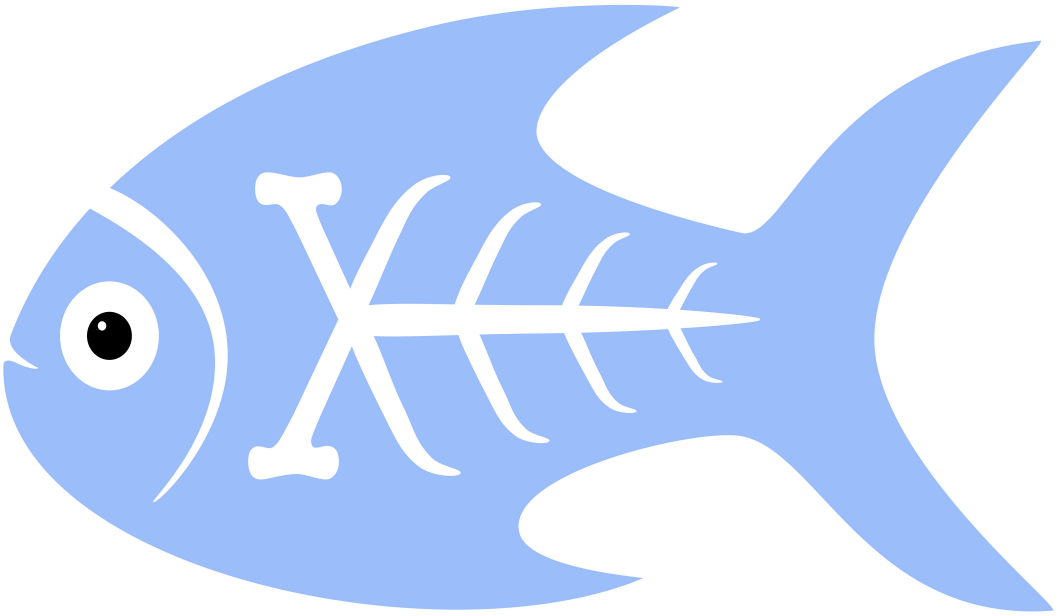
XProcRef

XProc 3.1 Step Reference



Erik Siegel - Xatapult
2025-06-25

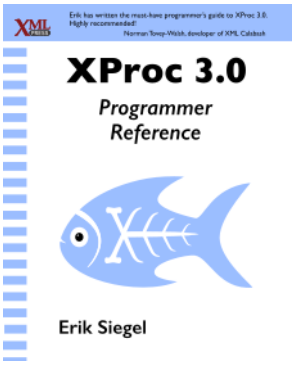
0 Table of Contents

1 Introduction

XProc is a programming language for processing XML, JSON, and other documents in pipelines. XProc chains conversions and other steps, allowing for potentially complex processing. XProc is especially useful for applications, such as publishing, where content may come from multiple input sources, pass through multiple processing steps and result in multiple output streams. More information, including lots of learning materials, can be found at https://xproc.org.

The basic building blocks of XProc are its *steps*. A step is something that processes the document(s) flowing through it in some way. For instance by changing some attributes, deleting stuff, or using it for accessing resources by HTTP. XProc has many built-in steps and you need to know, in detail, what they do to be able to write an XProc program. This book is a reference guide to all defined XProc steps.

In 2020 I published a book called "XProc 3.0 - Programmer Reference" (https://xmlpress.net/publications/xproc-3-0/):



Appendices A and B in the book describe all the steps. However, due to time constraints, the step descriptions were copied from the formal XProc specification. This leaves much to be desired for users of the language: the specification is aimed at XProc processor *implementers*, not at language *users*. To correct this, I created the website `xprocref.org` (https://xprocref.org), containing reference information about all the XProc steps, written from a more user-oriented perspective.

Also, since the book was written, the specification was updated to version 3.1. And although the changes are relative minor, details count! Both the `xprocref.org` (https://xprocref.org) website and this book are up-to-date with version 3.1.

Erik Siegel (erik@xatapult.nl)
Xatapult Content Engineering (https://www.xatapult.com)
2025-06-25

XATAPULT
CONTENT ENGINEERING

2 Steps

2.1 Overview

Steps for XProc version 3.1. You can also view these steps by category (pg. 207).

A

- `p:add-attribute` (pg. 6) - Add (or replace) an attribute on a set of elements.
- `p:add-xml-base` (pg. 8) - Add explicit xml:base attributes to a document.
- `p:archive` (pg. 11) - Perform operations on archive files.
- `p:archive-manifest` (pg. 18) - Create an XML manifest document describing the contents of an archive file.

C

- `p:cast-content-type` (pg. 22) - Changes the media type of a document.
- `p:compare` (pg. 29) - Compares documents for equality.
- `p:compress` (pg. 31) - Compresses a document.
- `p:count` (pg. 32) - Count the number of documents.
- `p:css-formatter` (pg. 34) - Renders a document using CSS formating.

D

- `p:delete` (pg. 36) - Delete nodes in documents.
- `p:directory-list` (pg. 37) - List the contents of a directory.

E

- `p:encode` (pg. 45) - Encodes a document.
- `p:error` (pg. 47) - Raises an error.

F

- `p:file-copy` (pg. 49) - Copies a file or directory.
- `p:file-create-tempfile` (pg. 51) - Creates a temporary file.
- `p:file-delete` (pg. 54) - Deletes a file or directory.
- `p:file-info` (pg. 56) - Returns information about a file or directory.
- `p:file-mkdir` (pg. 59) - Creates a directory.
- `p:file-move` (pg. 60) - Moves or renames a file or directory.
- `p:file-touch` (pg. 62) - Changes the modification timestamp of a file.
- `p:filter` (pg. 63) - Selects parts of a document.

H

- `p:hash` (pg. 65) - Computes a hash code for a value.
- `p:http-request` (pg. 68) - Interact using HTTP (or related protocols).

I

- `p:identity` (pg. 78) - Copies the source to the result without modifications.
- `p:insert` (pg. 80) - Inserts one document into another.
- `p:invisible-xml` (pg. 83) - Performs invisible XML processing.

J

L

M

N

O

P

R

S

T

U

- `p:unarchive` (pg. 147) - Extracts documents from an archive file.
- `p:uncompress` (pg. 152) - Uncompresses a document.
- `p:unwrap` (pg. 154) - Unwraps elements in a document.
- `p:uuid` (pg. 156) - Injects UUIDs into a document.

V

- `p:validate-with-dtd` (pg. 158) - Validates a document using a DTD.
- `p:validate-with-json-schema` (pg. 160) - Validates a JSON document using JSON schema.
- `p:validate-with-nvdl` (pg. 164) - Validate a document using NVDL.
- `p:validate-with-relax-ng` (pg. 167) - Validate a document using RELAX NG.
- `p:validate-with-schematron` (pg. 171) - Validates a document using Schematron.
- `p:validate-with-xml-schema` (pg. 176) - Validates a document using XML Schema.

W

- `p:wrap` (pg. 184) - Wraps nodes in a parent element.
- `p:wrap-sequence` (pg. 187) - Wraps a sequence of documents in an element.
- `p:www-form-urldecode` (pg. 189) - Decode a URL parameter string into a map.
- `p:www-form-urlencode` (pg. 191) - Encode parameters into a URL string.

X

- `p:xinclude` (pg. 193) - Apply XInclude procesing to a document.
- `p:xquery` (pg. 196) - Invoke an XQuery query.
- `p:xsl-formatter` (pg. 199) - Renders an XSL-FO document.
- `p:xslt` (pg. 200) - Invoke an XSLT stylesheet.

2.2 p:add-attribute

Add (or replace) an attribute on a set of elements.

Summary

```
<p:declare-step type="p:add-attribute">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="attribute-name" as="xs:QName" required="true"/>
  <option name="attribute-value" as="xs:string" required="true"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
</p:declare-step>
```

The `p:add-attribute` step adds (or replaces) an attribute. This is done for the element(s) matched by the `match` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to add (or replace) the attribute on.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`attribute-name`	`xs:QName`	`true`		The name of the attribute. This may contain a namespace specification.
`attribute-value`	`xs:string`	`true`		The value of the attribute.
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern that selects the element(s) to add (or replace) the attribute on. If not specified, the root element is used. This must be an XSLT match pattern that matches an element. If it matches any other kind of node, error `XC0023` (pg. 8) is raised.

Description

The `p:add-attribute` step:

- Takes the document appearing on its `source` port.
- Processes the elements that match the pattern in the `match` option:
  - If a selected element does *not* contain an attribute with the name given in the `attribute-name` option, an attribute with this name and a value as given in the `attribute-value` option is added to it.
  - If a selected element already has such an attribute, its value is replaced with the value given in the `attribute-value` option.
- The resulting document appears on its `result` port.

The `p:add-attribute` step can only set a *single* attribute. The `p:set-attributes` (pg. 124) step can be used to set multiple attributes at once.

Examples

Adding/replacing an attribute

This example adds a `type="special"` attribute to all `<text>` elements. One of the input `<text>` elements already has such an attribute, but with a different value. This existing attribute is replaced.

Source document:

```
<texts>
   <text>Hello there!</text>
   <text>This is funny…</text>
   <text type="normal">And that's normal.</text>
</texts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:add-attribute match="text" attribute-name="type" attribute-value="special"/>

</p:declare-step>
```

Result document:

```
<texts>
   <text type="special">Hello there!</text>
   <text type="special">This is funny…</text>
   <text type="special">And that's normal.</text>
</texts>
```

Additional details

- `p:add-attribute` preserves all document-properties of the document(s) appearing on its `source` port.
- If an attribute called `xml:base` is added or changed, the base URI of the element is updated accordingly. See also category Base URI related (pg. 212).
- You cannot use this step to add or change a namespace declaration. Attempting to do so will result in error `XC0059` (pg. 8).

  Note, however, that it is possible to add an attribute whose namespace is not in scope on the element it is added to. The XProc namespace fixup mechanism will take care of handling this and add the appropriate namespace declarations.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a node which is not an element.
`XC0059` (pg. 216)	It is a dynamic error if the QName value in the `attribute-name` option uses the prefix "`xmlns`" or any other prefix that resolves to the namespace name `http://www.w3.org/2000/xmlns/`.

2.3 p:add-xml-base

Add explicit xml:base attributes to a document.

Summary

```
<p:declare-step type="p:add-xml-base">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="all" as="xs:boolean" required="false" select="false()"/>
  <option name="relative" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:add-xml-base` step adds explicit `xml:base` attributes to the source document. An `xml:base` attribute annotates an element with a "base URI" value, which is usually the URI this element came from.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to add the `xml:base` attributes to.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`all`	`xs:boolean`	`false`	`false`	Whether to add the `xml:base` attributes to *all* elements.
`relative`	`xs:boolean`	`false`	`true`	Whether to make the value of the `xml:base` attributes on child elements of the root relative instead of absolute.

Description

Nodes (elements, attributes, etc.) in XML documents "remember" where they came from: they have a so-called "base URI" attached. This is usually the URI this node came from: the path (for instance on disk) to the document where it belonged to. In most cases the base URI is invisible. The `p:add-xml-base` step exposes this by adding (or adapting) `xml:base` attributes.

The operation of the `p:add-xml-base` step depends on the values of the `all` and `relative` options:

all	relative	Operation
false	false	• The root element of the document gets an `xml:base` attribute with an *absolute* URI. An existing `xml:base` attribute is updated. • Child elements where the base URI differs from its parent get an `xml:base` attribute with an *absolute* URI. An existing `xml:base` attribute is updated. • Any other `xml:base` attributes are removed.
false	true	These are the default values for these options. • The root element of the document gets an `xml:base` attribute with an *absolute* URI. An existing `xml:base` attribute is updated. • Child elements where the base URI differs from its parent get an `xml:base` attribute with an *relative* URI. An existing `xml:base` attribute is updated. • Any other `xml:base` attributes are removed.
true	false	• The root element of the document gets an `xml:base` attribute with an *absolute* URI. An existing `xml:base` attribute is updated. • *All* child elements get an `xml:base` attribute with an *absolute* URI. An existing `xml:base` attribute is updated.
true	true	This is not allowed. Error `XC0058` (pg. 11) is raised.

Examples

Straight step usage

The following example shows what happens if we use `p:add-xml-base` straight out of the box:
Assume the following source document called `in1.xml`:

```
<texts>
   <text>Hello XProc lovers…</text>
</texts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:add-xml-base/>

</p:declare-step>
```

Result document:

```
<texts xml:base="file:/…/…/in1.xml">
   <text>Hello XProc lovers…</text>
</texts>
```

We can create an `xml:base` attribute on *every* element by setting the `all` option to `true`. Because the default value for the `relative` option is `true` and both options can't both be `true`, we have to set the `relative` option to `false` explicitly.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:add-xml-base all="true" relative="false"/>

</p:declare-step>
```

Result document:

```
<texts xml:base="file:/…/…/in1.xml">
   <text xml:base="file:/…/…/in1.xml">Hello XProc lovers…</text>
</texts>
```

Multiple base URIs

The following example shows what happens if you combine two documents. For this, we're going to use another XML document called `in2.xml`:

```
<specialtext>Are you in for something special?</specialtext>
```

The following pipeline first creates a combined document by inserting this into the same source document we used in the previous example Straight step usage (pg. 9):

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert match="/*" position="last-child">
    <p:with-input port="insertion" href="in2.xml"/>
  </p:insert>

  <p:add-xml-base/>

</p:declare-step>
```

The result document has a relative `xml:base` value on the inserted document because the default value for the `relative` option is `true`:

```
<texts xml:base="file:/…/…/in1.xml">
   <text>Hello XProc lovers…</text>
   <specialtext xml:base="in2.xml">Are you in for something special?</specialtext>
</texts>
```

But we could also ask for absolute base URI values:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert match="/*" position="last-child">
    <p:with-input port="insertion" href="in2.xml"/>
  </p:insert>

  <p:add-xml-base relative="false"/>

</p:declare-step>
```

Result document:

```
<texts xml:base="file:/…/…/in1.xml">
   <text>Hello XProc lovers…</text>
   <specialtext xml:base="file:/…/…/in2.xml">Are you in for something special?</specialtext>
</texts>
```

Additional details

- `p:add-xml-base` preserves all document-properties of the document(s) appearing on its `source` port.
- The `xml` namespace prefix as used here is bound to the namespace `http://www.w3.org/XML/1998/namespace`. In most cases you *don't* have to bind this prefix explicitly (by adding `xmlns:xml="http://www.w3.org/XML/1998/namespace"`). This namespace binding is part of the XML language.
- A formal definition of base URIs and `xml:base` attributes can be found here (https://www.w3.org/TR/xmlbase/).

Errors raised

Error code	Description
`XC0058` (pg. 216)	It is a dynamic error if the `all` and `relative` options are both `true`.

2.4 p:archive

Perform operations on archive files.

Summary

```
<p:declare-step type="p:archive">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <input port="archive" primary="false" content-types="any" sequence="true">
    <p:empty/>
  <input/>
  <input port="manifest" primary="false" content-types="xml" sequence="true">
    <p:empty/>
  <input/>
  <output port="report" primary="false" content-types="application/xml" sequence="false"/>
  <option name="format" as="xs:QName" required="false" select="'zip'"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="relative-to" as="xs:anyURI?" required="false" select="()"/>
</p:declare-step>
```

The `p:archive` step can perform several different operations on archive files (for instance ZIP files). The most common one will likely be creating one, but it could also provide services like update, freshen or even merge. The resulting archive appears on its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The `source` port is used to provide the documents to be archived. How and which of these documents are processed is governed by the document(s) appearing on the other input ports and the combination of options and parameters. See below for details.
`result`	`output`	`true`	`any`	`false`	The resulting archive.
`archive`	`input`	`false`	`any`	`true`	Optional archives for operations like update, freshen or merge.
`manifest`	`input`	`false`	`xml`	`true`	An optional manifest document that tells the step how to construct the archive. If no manifest document is provided on this port, a default manifest is constructed automatically. See "The XML archive manifest document format" on page 12 for details.
`report`	`output`	`false`	`application/xml`	`false`	A report about the archiving operation. This will be the same as the manifest, optionally amended with additional attributes and/or elements.

Options:

Name	Type	Req?	Default	Description
`format`	`xs:QName`	`false`	`zip`	The format of the archive. • If its value is `zip` (the default), the `p:archive` step expects a ZIP archive on the `source` port. • Whether any other archive formats can be handled and what their names (values for this option) are is implementation-defined and therefore dependent on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the archiving. Several parameters are defined for processing ZIP archives (see "Handling of ZIP archives" on page 14). A specific XProc processor might define its own.
`relative-to`	`xs:anyURI?`	`false`	`()`	This is option is used in creating a manifest when no manifest is provided on the `manifest` port. If a manifest is present this option is not used.

Description

The `p:archive` step is the Swiss army knife for handling archives. Its most common use is creating archives, but it could also be used for operations like update, freshen or even merge.

To make all this possible, the operation of `p:archive` is unfortunately quite complicated. The details are below, here's a summary:

- What's exactly in the resulting archive is controlled using a manifest document (see "The XML archive manifest document format" on page 12). In such a manifest you specify the URI of the document to add and the path of this document *in* the archive.

  A manifest of an existing archive, sometimes useful as a starting point, can be produced using the `p:archive-manifest` (pg. 18) step.
- Besides the documents in the manifest you can also specify documents to add by providing these on the step's `source` port. Any document appearing on this port that is not already mentioned in the manifest is automatically added to the manifest. The path of such a document *in* the resulting archive can be controlled using the `relative-to` option.
- When adding documents to the archive, `p:archive` compares the base URIs in the manifest with those of the documents appearing on the `source` port (the value of the `base-uri` document-property). If these match, the document on the `source` port is added. If not, the URI in the manifest is used to load a document (usually from disk).

Archives come in many formats. The only format the `p:archive` step is required to handle is ZIP. However, depending on the XProc processor used, other formats may also be processed.

The XML archive manifest document format

An archive manifest is an XML document that specifies files to process constructing the archive. It is also used as the result format of the `p:archive-manifest` (pg. 18) step.

Its root element is `<c:archive>` (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace):

```
<c:archive>
  ( <c:entry> |
    (any other element)
  )*
</c:archive>
```

Child element	#	Description
`c:entry`	*	An entry (a file) in the archive.

A `<c:entry>` element describes a single entry (a file) in the archive:

```
<c:entry (any other attribute)
        href = xs:anyURI
        name = xs:string
        comment? = xs:string
        content-type? = xs:string
        level? = xs:string
        method? = xs:string >
  (any child element)*
</c:entry>
```

Attribute	#	Type	Description
`href`	1	`xs:anyURI`	The URI of the entry. This plays an important role in determining which and how files are added to the archive, see below. A relative value is made absolute against the base URI of the manifest itself.
`name`	1	`xs:string`	The name of the entry. This is the path of the file *within* the archive. Usually this is a relative path. However, depending on how archives are constructed, an absolute path (a path starting with a `/`) is possible. Archives constructed by XProc steps always produce relative paths (no leading `/`).
`comment`	?	`xs:string`	An optional comment associated with the entry.
`content-type`	?	`xs:string`	The content-type (MIME type) of the entry. The `p:archive` step ignores it, but the `p:archive-manifest` step always adds it.
`level`	?	`xs:string`	The compression level of the entry. There are no defined values, all values are XProc processor dependent.
`method`	?	`xs:string`	The compression method of the entry. There is only one defined value: `none`, meaning, of course, no compression. Any other values are XProc processor dependent.

The p:archive algorithm

The `p:archive` step follows a, rather complicated, algorithm. It has two phases:

1 - Construct a complete manifest

First, the manifest (the document, if any, appearing on the `manifest` port) is checked and completed if necessary:

- If no document appears on the `manifest` port, an empty manifest is created.
- The base URIs of the documents appearing on the `source` port are compared against the list of base URIs in the manifest (the `c:entry/@href` values, made absolute). If there are documents on the source port that are *not* in the manifest, an entry (`<c:entry>` element) for this document is created:
  - The `c:entry/@href` attribute becomes the base URI of the document.
  - The `c:entry/@name` (which is the path/name of the entry in the archive) is computed against the value of the `relative-to` option:
    - If the base URI of the document starts with the value of the `relative-to` option, the `c:entry/@name` attribute value becomes the substring after this.
    - If the base URI of the document does not start with the value of the `relative-to` option, the `c:entry/@name` attribute value becomes the path of this base URI (without a leading `/`).

    For instance, assume the `relative-to` option is set to `file:///some/path/`. A document with base URI `file:///some/path/etc/x.txt` gets a `c:entry/@name` attribute value `etc/x.txt`. A document with base URI `file:///someother/path/y.txt` gets a `c:entry/@name` attribute value `someother/path/y.txt`.

The result of all this is that we now have a manifest that has entries (`<c:entry>` elements) for all documents appearing on the `source` port. It can also have entries for documents that are *not* on the source port: because such an entry was present in the initial manifest and no matching document on the `source` port was found for it.

2 - Process the manifest

The now completed manifest is processed. For every entry (`<c:entry>` element):

- If the value of the `c:entry/@href` attribute matches the base URI of one of the documents appearing on the `source` port, this document is added to the archive.

  When appropriate (for instance for XML documents), the value of its (optional) `serialization` document-property is used for serializing it (convert it to text format).
- For other entries, the value of the `c:entry/@href` attribute is used to load the file (for instance from disk if it starts with `file:/`) and add it to the archive.

  These documents are used "as is": no parsing/serialization takes place.

In both cases, the value of the `c:entry/@name` attribute becomes the name/path of the entry in the archive. The values of the other attributes of the `<c:entry>` element might also get used, but this is dependent on the XProc processor used and/or the archive's format.

The `p:archive` step is supposed to retain the order of the `<c:entry>` elements. This is, for instance, important when constructing an e-book in EPUB format: this has a non-compressed entry that must be first in the archive.

Handling of ZIP archives

When the value of the `format` option is absent or `zip`, the following applies:

- The values of the `c:entry/@name` attributes in the manifest must be relative paths (without a leading `/`).
- The `archive` port accepts zero or one ZIP archive. If this port is empty, an empty ZIP archive is used as its default value.
- The `parameters` option is a map that associates parameters (the keys in the map) with values. For ZIP archives, the following parameters can be used:

Parameter	Description
`command`	Specifies the operation to perform. It's default value is `update`. See below for a description of the commands.
`level`	For entries that have no `c:entry/@level` attribute specified, this is the default compression level for entries added or updated in the archive. For ZIP archives, its possible values are: • `smallest` • `fastest` • `default` • `huffman` • `none`
`method`	For entries that have no `c:entry/@method` attribute specified, this is the default compression method for entries added or updated in the archive. For ZIP archives, its possible values are: • `deflated` • `none`

  The `command` parameter can have one of the following values:

Command	Description
`update` (default)	The archive appearing on the `archive` port is updated: • An entry in this ZIP archive that corresponds with a `c:entry/@name` attribute in the manifest gets updated as specified in the `<c:entry>` element. • For other entries in the ZIP archive, first their name/path is made absolute using the base URI of the archive. If a file exists with that URI and is newer than the entry in the ZIP archive, it is updated. • For all `<c:entry>` elements in the manifest that have no corresponding entry in the ZIP archive, the document gets added. Please note that when there is no document on the `archive` port, `p:archive` will always create a new, fresh, archive.
`create`	This behaves like the `update` command except that timestamps are ignored and updates (if any) always take place.
`freshen`	This behaves like the `update` command except that no new files will be added.
`delete`	For the `delete` command a ZIP archive `must` be present on the `archive` port. It removes all entries in the ZIP archive that have a corresponding `c:entry/@name` attribute in the manifest. All other manifest entries are ignored.

Examples

Basic usage

In probably most cases, the `p:archive` step will be used to create an archive. If you have no special requirements this is easy: simply supply the documents for the archive on the step's `source` port. The only thing you need to take into account is the name/path of the entries in the archive: for this the `relative-to` option is important.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="test/in2.xml"/>
  </p:input>
  <p:output port="result"/>

  <p:variable name="relative-to" select="resolve-uri('.', static-base-uri())"/>

  <p:archive relative-to="{$relative-to}"/>

  <p:store href="tmp/result.zip"/>
  <p:archive-manifest relative-to="{$relative-to}"/>

</p:declare-step>
```

Result document:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry name="in1.xml"
            content-type="application/xml"
            href="file:/…/…/in1.xml"
            method="deflated"
            size="91"
            compressed-size="80"
            time="2025-06-25T13:24:50+02:00"/>
   <c:entry name="test/in2.xml"
            content-type="application/xml"
            href="file:/…/…/test/in2.xml"
            method="deflated"
            size="98"
            compressed-size="84"
            time="2025-06-25T13:24:50+02:00"/>
</c:archive>
```

- The pipeline's input consists of two documents, `in1.xml` and `test/in2.xml`. Note that (because the `p:document/@href` attributes have relative values) the paths to these documents are relative to the location of the pipeline itself.
- When we construct an archive we usually don't want the full path of the files on disk in the archive also. In this case we choose to use their relative paths against the pipeline. To achieve this we need the path (directory) where the pipeline is stored. This is done with the expression `resolve-uri('.', static-base-uri())` and stored in the `relative-to` variable.
- We then create the archive using `p:archive`. The two input documents appear on its `source` port. We do not provide a manifest on the `manifest` port, so one will get constructed automatically.
- The names of the entries in the resulting archive get constructed by "subtracting" the value of the `relative-to` option from the base URIs of the source documents. The results will be their relative names against the pipeline's location.
- We store the resulting zip and, just to show you what's inside, ask for an archive manifest using the `p:archive-manifest` (pg. 18) step.

Using the report port

The `p:archive` step also has a `report` port that outputs the manifest of the resulting archive. So, building on the Basic usage (pg. 15) example, we could also have shown what's inside the created archive like this:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="test/in2.xml"/>
  </p:input>
  <p:output port="result" pipe="report@create-archive"/>

  <p:variable name="relative-to" select="resolve-uri('.', static-base-uri())"/>

  <p:archive relative-to="{$relative-to}" name="create-archive"/>

  <p:store href="tmp/result.zip"/>

</p:declare-step>
```

Result document:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry href="file:/…/…/in1.xml" name="in1.xml"/>
   <c:entry href="file:/…/…/test/in2.xml" name="test/in2.xml"/>
</c:archive>
```

Note that the information in the manifest is less than what `p:archive-manifest` (pg. 18) produces. What exactly happens here is implementation-defined and therefore dependent on the XProc processor used.

Using a manifest

This example creates a manifest that references some additional file for the archive. Note that in the archive we give it a different name than its source using the `c:entry/@name` attribute. When the manifest is processed, `p:archive` notices that `test/in2.xml` is not on its `source` port and therefore loads it from disk.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0" name="example">

  <p:input port="source" href="in1.xml"/>
  <p:output port="result"/>

  <p:identity name="manifest">
    <p:with-input>
      <c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
        <c:entry name="test/extra.xml" href="test/in2.xml"/>
      </c:archive>
    </p:with-input>
  </p:identity>

  <p:variable name="relative-to" select="resolve-uri('.', static-base-uri())"/>
  <p:archive relative-to="{$relative-to}">
    <p:with-input pipe="source@example"/>
    <p:with-input port="manifest" pipe="result@manifest"/>
  </p:archive>

  <p:store href="tmp/result.zip"/>
  <p:archive-manifest relative-to="{$relative-to}"/>

</p:declare-step>
```

Result document:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry name="test/extra.xml"
            content-type="application/xml"
            href="file:/…/…/test/extra.xml"
            method="deflated"
            size="62"
            compressed-size="49"
            time="2025-02-05T13:05:59+01:00"/>
   <c:entry name="in1.xml"
            content-type="application/xml"
            href="file:/…/…/in1.xml"
            method="deflated"
            size="91"
            compressed-size="80"
            time="2025-06-25T13:24:50+02:00"/>
</c:archive>
```

Additional details

- The only document-property for the document appearing on the `result` port is `content-type` (its value depending on the archive's format). Note it has no `base-uri` document-property and no document-properties from the document on the `source` or `archive` port survive.
- Documents appearing on the `source` port must have a `base-uri` document-property. All these `base-uri` document-properties must have a unique value.
- A relative value for the `relative-to` option gets de-referenced against the base URI of the element in the pipeline it is specified on. In most cases this will be the path of the pipeline document.
- The only format this step is required to handle is ZIP. The ZIP format definition can be found here (https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT).

Errors raised

Error code	Description
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XC0080` (pg. 217)	It is a dynamic error if the number of documents on the `archive` does not match the expected number of archive input documents for the given `format` and `command`.
`XC0081` (pg. 217)	It is a dynamic error if the format of the archive does not match the format as specified in the `format` option.
`XC0084` (pg. 217)	It is a dynamic error if two or more documents appear on the `p:archive` step's `source` port that have the same base URI or if any document that appears on the `source` port has no base URI.
`XC0085` (pg. 217)	It is a dynamic error if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.

Error code	Description
`XC0100` (pg. 217)	It is a dynamic error if the document on port `manifest` does not conform to the given schema.
`XC0112` (pg. 218)	It is a dynamic error if more than one document appears on the port `manifest`.
`XC0118` (pg. 218)	It is a dynamic error if an archive manifest is invalid according to the specification.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.5 p:archive-manifest

Create an XML manifest document describing the contents of an archive file.

Summary

```
<p:declare-step type="p:archive-manifest">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="format" as="xs:QName?" required="false" select="()"/>
  <option name="override-content-types" as="array(array(xs:string))?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="relative-to" as="xs:anyURI?" required="false" select="()"/>
</p:declare-step>
```

The `p:archive-manifest` step creates an XML manifest document describing the contents of the archive file appearing on its `source` port (for instance a ZIP file).

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The archive file to create the manifest for.
`result`	`output`	`true`	`application/xml`	`false`	The created XML manifest document. See the `p:archive` (pg. 11) step for a description of its format.

Options:

Name	Type	Req?	Default	Description
`format`	`xs:QName?`	`false`	`()`	The format of the archive file on the `source` port: • If its value is `zip`, the `p:archive-manifest` step expects a ZIP archive on the `source` port. • If absent or the empty sequence, the `p:archive-manifest` step tries to guess the archive file format. The only format that this step is required to recognize and handle is ZIP. • Whether any other archive formats can be handled and what their names (values for this option) are depends on the XProc processor used.
`override-content-types`	`array(array (xs:string))?`	`false`	`()`	Use this to override the content-type determination of the files in the archive (see "Overriding content-types" on page 20).
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the XML manifest document generation. The XProc specification does not define any parameters for this option. A specific XProc processor might define its own.
`relative-to`	`xs:anyURI?`	`false`	`()`	This option can be used to set/override the base URI of the archive. If you don't specify it, it is, as expected, the base URI of the document appearing on the `source` port. The use of this option is rare, but you might need it when: • The archive on the `source` port has no `base-uri` document-property. This would raise error `XC0120` (pg. 22). • You use this manifest as a base for creating a new one with `p:archive` (pg. 11). The base URI plays an important role here and setting it to specific value is sometimes useful.

Description

The `p:archive-manifest` step takes an archive file (for instance a ZIP file) on its `source` port and returns on its `result` port an XML document describing the contents of the archive: the *archive manifest*. The archive manifest format is described in the `p:archive` (pg. 11) step.

Archive manifests can be used in several ways. Some examples:

- To inspect which files are present in an archive, for instance to check whether what you've got is complete.
- As an input manifest for `p:archive` (pg. 11). This step takes, on its `manifest` port, a manifest like the one produced by `p:archive-manifest` and uses this to create a new archive or update an existing one. You could for instance first get a manifest using `p:archive-manifest`, change it to reflect the changes you need and then feed it to `p:archive` (pg. 11) to produce a new archive.

Archives come in many formats. The only format the `p:archive-manifest` step is required to handle is ZIP. However, depending on the XProc processor used, other formats may also be processed.

Overriding content-types

One of the things the `p:archive-manifest` step does is determining the content-type (MIME type) of the archive entries. This is usually done based on the filename/extension. It is recorded in the manifest `c:entry/@content-type` attribute.

Sometimes it is useful to override this mechanism and assign specific content-types to some of the entries. For instance, the files Microsoft Office produces (`.docx`, `.xlsx`, etc.) are archives with a lot of XML documents inside. Some of these documents have the extension `.rels` and would therefore not be recognized as XML documents. The `override-content-types` option makes it possible to adjust this behavior.

The value of the `override-content-types` option must be an array of arrays. The inner arrays must have exactly two members:

- The first member must be an XPath regular expression.
- The second member must be a valid a MIME content-type.

Determining an archive entry's content-type is now as follows:

- The inner arrays of the `override-content-types` option value are processed in order of appearance (so order is significant).
- The XPath regular expression (in the first member of the inner array) is matched against the full path of an entry *in* the archive (as in `matches($path-in-archive, $regular-expression)`).
- If a match is found, the content-type (the second member of the inner array) is used as the entry's content-type.
- If no match was found for all the inner arrays, the normal mechanism for determining the content-type is used.

For example: setting the `override-content-types` option to `[ ['.rels$', 'application/xml'], ['^special/', 'application/octet-stream'] ]` means that all files ending with `.rels` will get the content-type `application/xml`. All files in the archive's `special` directory (including sub-directories) will get the content-type `application/octet-stream`. See also the Overriding content types (pg. 21) example.

Examples

Basic usage

Assume we have a simple ZIP archive with two entries:

- An XML file in the root called `reference.xml`
- An image in an `images/` sub-directory called `logo.png`.

The following pipeline creates an archive manifest for this ZIP file:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:archive-manifest/>

</p:declare-step>
```

Resulting archive manifest:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry name="images/logo.png"
            content-type="image/png"
            href="file:/…/…/test.zip/images/logo.png"
            method="deflated"
            size="86656"
            compressed-size="85694"
            time="2024-07-04T11:12:22.4+02:00"/>
   <c:entry name="reference.xml"
            content-type="application/xml"
            href="file:/…/…/test.zip/reference.xml"
            method="deflated"
            size="78"
            compressed-size="77"
            time="2024-07-09T19:58:50.75+02:00"/>
</c:archive>
```

As you can see, the XProc processor I'm using to process this example (MorganaXProc-III) adds a few extra attributes to the `<c:entry>` elements: `size`, `compressed-size` and `time`.

Also note the contents of the `c:entry/@href` attributes: they are a combination of the full path/filename of the archive and the path of the entry within the archive (as in the `c:entry/@name` attribute). The `c:entry/@href` attribute plays an important role when creating archives using `p:archive` (pg. 11).

Overriding content types

This example uses the same ZIP archive as in Basic usage (pg. 20). The following pipeline explicitly sets the content type for `.png` files to `application/octet-stream`:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:archive-manifest>
    <p:with-option name="override-content-types" select="[ ['\.png$', 'application/octet-stream'] ]"/>
  </p:archive-manifest>

</p:declare-step>
```

Resulting archive manifest:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry name="images/logo.png"
            content-type="application/octet-stream"
            href="file:/…/…/test.zip/images/logo.png"
            method="deflated"
            size="86656"
            compressed-size="85694"
            time="2024-07-04T11:12:22.4+02:00"/>
   <c:entry name="reference.xml"
            content-type="application/xml"
            href="file:/…/…/test.zip/reference.xml"
            method="deflated"
            size="78"
            compressed-size="77"
            time="2024-07-09T19:58:50.75+02:00"/>
</c:archive>
```

Using the relative-to option

This example uses the same ZIP archive as in Basic usage (pg. 20). It sets the `relative-to` to `file:///test/`. This is reflected in the `c:entry/@href` attributes:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:archive-manifest relative-to="file:///test/"> </p:archive-manifest>

</p:declare-step>
```

Resulting archive manifest:

```
<c:archive xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:entry name="images/logo.png"
            content-type="image/png"
            href="file:///test/images/logo.png"
            method="deflated"
            size="86656"
            compressed-size="85694"
            time="2024-07-04T11:12:22.4+02:00"/>
   <c:entry name="reference.xml"
            content-type="application/xml"
            href="file:///test/reference.xml"
            method="deflated"
            size="78"
            compressed-size="77"
            time="2024-07-09T19:58:50.75+02:00"/>
</c:archive>
```

Additional details

- The only document-property for the document appearing on the `result` is `content-type`, with value `application/xml`. Note it has no `base-uri` document-property and no document-properties from the document on the `source` port survive.
- A relative value for the `relative-to` option gets de-referenced against the base URI of the element in the pipeline it is specified on. In most cases this will be the path of the pipeline document.
- The only format this step is required to handle is ZIP. The ZIP format definition can be found here (https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT).

Errors raised

Error code	Description
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XC0085` (pg. 217)	It is a dynamic error if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.
`XC0120` (pg. 218)	It is a dynamic error if the `relative-to` option is not present and the document on the `source` port does not have a base URI.
`XC0146` (pg. 219)	It is a dynamic error if the specified value for the `override-content-types` option is not an array of arrays, where the inner arrays have exactly two members of type `xs:string`.
`XC0147` (pg. 219)	It is a dynamic error if the specified value is not a valid XPath regular expression.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.6 p:cast-content-type

Changes the media type of a document.

Summary

```
<p:declare-step type="p:cast-content-type">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="content-type" as="xs:string" required="true"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:cast-content-type` step takes the document appearing on its `source` port and changes its media type according to the value of the `content-type` option, transforming the document if necessary.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The document to change the media type of.
`result`	`output`	`true`	`any`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`content-type`	`xs:string`	`true`		The media type of the resulting document. This must be a valid media type (either `type/subtype` or `type/subtype+ext`). If not, error `XD0079` (pg. 29) is raised.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the casting/transformation of the document. Keys, values and their meaning are dependent on the XProc processor used.

Description

A document flowing through an XProc pipeline has a *media type*, which tells the XProc processor what kind of document it is dealing with. The media type of a document is recorded in its `content-type` document-property. Example values are `text/xml` for XML documents, `application/json` for JSON documents, etc. For more information about media types see for example Wikipedia (https://en.wikipedia.org/wiki/Media_type).

The `p:cast-content-type` step has a required `content-type` option and tries to cast (change) the media type of the document appearing on its `source` port according to the value of this option. Sometimes this is a (very) simple operation: for instance, changing one XML media type to another just changes the value of the `content-type` document-property. However, you can also request more complex changes, like converting an XML document into JSON or vice versa.

Of course, not every media type can be cast into every other media type. The following sections describe what you can (and cannot) do. If you request an impossible cast, error `XC0071` (pg. 28) is raised.

A brief explanation of media types and how XProc treats them can be found in the "XProc media type usage" on page 25 section below.

Converting XML documents

When the input document is an XML document (has an XML media type), the following casts are supported:

- Casting to another XML media type simply changes the `content-type` document-property.
- Casting to an HTML media type changes the `content-type` document-property and removes any `serialization` document-property.
- Casting to a JSON media type converts the XML into JSON:
  - The XPath and XQuery Functions and Operators 3.1 (https://www.w3.org/TR/xpath-functions-31) standard defines an XML format for the representation of JSON data (https://www.w3.org/TR/xpath-functions-31/#json-to-xml-mapping). The XPath function `xml-to-json()` (https://www.w3.org/TR/xpath-functions-31/#func-xml-to-json) converts this format into a JSON conformant string (and for further processing, `parse-json()` (https://www.w3.org/TR/xpath-functions-31/#func-parse-json) turns this string into a map/array).

    If an input document of `p:cast-content-type` is conformant to this XML format for the representation of JSON data (https://www.w3.org/TR/xpath-functions-31/#json-to-xml-mapping), it's converted into its JSON equivalent (like calling `parse-json(xml-to-json())`). See Converting the XML representation of JSON (pg. 26) for an example.
  - If the input document has a `<c:param-set>` root element and `<c:param name="…" value="…"/>` child elements (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace), it will turn this into a JSON map with the values of the `name` attributes as keys. See the Converting param-sets (pg. 26) example.

Param-sets are an XProc 1.0 construct, used for passing parameters (there were no maps in those days). Unless you're converting XProc 1.0 steps into 3.x, it's unlikely you'll need this feature.

- In all other cases it's up to the XProc processor what happens. It could turn your XML into some kind of JSON, but it could just as well raise an error.

A `serialization` document-property is removed when converting to JSON.

- Casting to a text media type converts the XML into text. The incoming XML comes out as text, as a string, complete with tags, attributes, etc.

  The result of this conversion is the same as calling the XPath `serialize($doc, $param)` (https://www.w3.org/TR/xpath-functions-31/#func-serialize) function, where `$doc` is the document to convert and `$param` is its `serialization` document-property. See the Converting XML to text (pg. 27) example.

  A `serialization` document-property is removed.
- Casting to any other media type where the input document is a `<c:data>` document (see "c:data documents" on page 25) results in a document with the specified media type and a representation that is the content of the `<c:data>` element after decoding it. The value of the `c:data/@content-type` attribute and the value of the `content-type` option of `p:cast-content-type` must be the same!

  A `serialization` document-property is removed.
- Casting to any other media type where the input is not a valid `<c:data>` document is implementation-defined and therefore dependent on the XProc processor used.

Converting HTML documents

When the input document is an HTML document (has an HTML media type), the following casts are supported:

- Casting to another HTML media type simply changes the `content-type` document-property.
- Casting to an XML media type changes the `content-type` document-property and removes a `serialization` document-property.
- Casting to a JSON media type is implementation-defined and therefore dependent on the XProc processor used.
- Casting to a text media type works the same as casting an XML media type to text. See "casting XML to text" on page 24 above.
- Casting to any other media type is implementation-defined and therefore dependent on the XProc processor used.

Converting JSON documents

When the input document is a JSON document (has a JSON media type), the following casts are supported:

- Casting to another JSON media type simply changes the `content-type` document-property.
- Casting to an HTML media type is implementation-defined and therefore dependent on the XProc processor used.
- Casting to an XML media type converts the JSON into XML according to the rules specified in the XPath XML format for the representation of JSON data (https://www.w3.org/TR/xpath-functions-31/#json-to-xml-mapping). See the Converting JSON into XML (pg. 27) example.

  A `serialization` document-property is removed.
- Casting to a text media type converts the JSON into text. The incoming JSON (which in XProc consists of maps/arrays) comes out as text, as a string.

  The result of this conversion is the same as calling the XPath `serialize($doc, $param)` (https://www.w3.org/TR/xpath-functions-31/#func-serialize) function, where `$doc` is the document to convert and `$param` is its `serialization` document-property.

  A `serialization` document-property is removed.
- Casting to any other media type is implementation-defined and therefore dependent on the XProc processor used.

Converting text documents

When the input document is an text document (has a text media type), the following casts are supported:

- Casting to another text media type simply changes the `content-type` document-property.
- Casting to an XML media type parses the text value of the document by calling the XPath `parse-xml()` (https://www.w3.org/TR/xpath-functions-31/#func-parse-xml) function. This assumes of course that the text is a well-formed XML document. If not, error `XD0049` (pg. 28) is raised.
- Casting to an HTML media type parses the document into an HTML document. How this is done is implementation-defined and therefore dependent on the XProc processor used. If unsuccessful, error `XD0060` (pg. 29) is raised.
- Casting to a JSON media type parses the document by calling the XPath `parse-json($doc, $param)` (https://www.w3.org/TR/xpath-functions-31/#func-parse-json) function, where `$doc` is the document to convert and `$param` is its `serialization` document-property.

  A `serialization` document-property is removed.
- Casting to any other media type is implementation-defined and therefore dependent on the XProc processor used.

Converting other media types

When the input document has any other media type (meaning XProc treats it as a binary document), the following casts are supported:

- Casting from an unrecognized media type to an XML media type produces a `<c:data>` document (see "c:data documents" on page 25). The `<c:data/@content-type>` attribute is the document's content type. The content of the `c:data` element is the base64 encoded representation of the document. See the Converting a binary media type into XML (pg. 28) example.

  A `serialization` document-property is removed.
- Casting from an unrecognized media type to a HTML, JSON, text or other unrecognized media type is implementation-defined and therefore dependent on the XProc processor used.

c:data documents

The `p:cast-content-type` step uses `<c:data>` documents to convert XML from and into binary media types (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace):

```
<c:data content-type = xs:string
        charset? = xs:string
        encoding? = xs:string />
```

Attribute	#	Type	Description
`content-type`	1	`xs:string`	The MIME type of the content.
`charset`	?	`xs:string`	The character set of the content, for instance `UTF-8` or `ASCII`. For an explanation of character encodings see Wikipedia.
`encoding`	?	`xs:string`	The encoding of the content. The most used encoding is `base64` (see Wikipedia).

XProc media type usage

A document *media type* (in XProc passed around in the `content-type` document-property) tells XProc (and your code if it needs to know this) what kind of document we're dealing with: the *document type*. XProc recognizes and handles five document types: XML, HTML, JSON, text and binary.

The relation between document type and media type is as follows:

Document type	Media types	Examples

XML	`*/xml`
	`*/*+xml` except `application/xhtml+xml`
		`text/xml`
		`application/xml`
		`image/svg+xml`

HTML	`text/html` `application/xhtml+xml`	`text/html` `application/xhtml+xml`

Document type	Media types	Examples
JSON	`application/json`	`application/json`
Text	`text/*` (not matching one of the XML or HTML media types)	`text/plain` `text/csv`

Binary	Anything else	`image/jpeg`
		`application/octet-stream`
		`application/zip`

Examples

Converting the XML representation of JSON

If an input document of `p:cast-content-type` is conformant to the XPath XML format for the representation of JSON data (https://www.w3.org/TR/xpath-functions-31/#json-to-xml-mapping) and the `content-type` option is a JSON media type, `p:cast-content-type` converts this into its JSON equivalent. The following source document is a shortened version of the example in the XPath standard:

```
<map xmlns="http://www.w3.org/2005/xpath-functions">
   <string key="desc">Distances </string>
   <boolean key="uptodate">true</boolean>
   <null key="author"/>
   <map key="cities">
      <array key="Brussels">
         <map>
            <string key="to">London</string>
            <number key="distance">322</number>
         </map>
      </array>
   </map>
</map>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:cast-content-type content-type="application/json"/>

</p:declare-step>
```

The resulting JSON map:

```
{"desc":"Distances ","uptodate":true,"author":null,"cities":{"Brussels":[{"to":"London","distance":322}]}}
```

Converting param-sets

Param-sets are constructs used in the XProc 1.0 days for passing sets of parameters, for instance to XSLT stylesheets. The current version uses maps for this. To enable converting param-sets into maps, `p:cast-content-type` contains support for this. In XProc, a map is JSON data, so the `content-type` option must be a JSON media type.

The source param-set document:

```
<c:param-set xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:param name="param1" value="y"/>
   <c:param name="param2" value="1234"/>
</c:param-set>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:cast-content-type content-type="application/json"/>

</p:declare-step>
```

The resulting JSON map:

```
{"param1":"y","param2":"1234"}
```

JSON maps are passed around as XPath maps, so it's easy to store such a map in a variable and use it later. Just add the following variable declaration directly after the `p:cast-content-type` invocation:

```
<p:variable name="param-set-map" as="map(*)" select="."/>
```

Unless you're converting XProc 1.0 code into a newer version, i's unlikely you'll need this param-set conversion feature.

Converting XML to text

Let's convert this simple XML document into text:

```
<input-document timestamp="2024-08-23T09:12:45">
   <text color="red">Hi there!</text>
</input-document>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:cast-content-type content-type="text/plain"/>

</p:declare-step>
```

The resulting *text* (it looks like it is another XML document, but it is just text):

```
<?xml version="1.0" encoding="utf-8"?><input-document timestamp="2024-08-23T09:12:45">
      <text color="red">Hi there!</text>
   </input-document>
```

Now assume we need this text representation without the XML header (the `<?xml … ?>` part at the top). The `p:cast-content-type` step uses the document `serialization` document-property to guide the conversions. This document-property is a map containing the required serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/). For this example: `map{'omit-xml-declaration': true()}`.

Document-properties can be specified using the `p:set-properties` (pg. 126) step. The value of the `properties` option of `p:set-properties` is itself a map, with the document-property names as keys. Therefore, its value becomes a map within a map: `map{'serialization': map{'omit-xml-declaration': true()}}`.

The following code (using the same input document as above) does the trick:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:set-properties properties="map{'serialization': map{'omit-xml-declaration': true()}}"/>

  <p:cast-content-type content-type="text/plain"/>

</p:declare-step>
```

Result document:

```
<input-document timestamp="2024-08-23T09:12:45">
      <text color="red">Hi there!</text>
   </input-document>
```

Converting JSON into XML

Converting JSON into XML means `p:cast-content-type` produces XML according to the XPath XML format for the representation of JSON data (https://www.w3.org/TR/xpath-functions-31/#json-to-xml-mapping) specification. Here we do the inverse of what is done in the Converting the XML representation of JSON (pg. 26) example.

Source document:

```
{"desc":"Distances","uptodate":true,"author":null,"cities":{"Brussels":[{"to":"London","distance":322}]}}
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:cast-content-type content-type="text/xml"/>

</p:declare-step>
```

Result document:

```
<map xmlns="http://www.w3.org/2005/xpath-functions">
   <string key="desc">Distances</string>
   <boolean key="uptodate">true</boolean>
   <null key="author"/>
   <map key="cities">
      <array key="Brussels">
         <map>
            <string key="to">London</string>
            <number key="distance">322</number>
         </map>
      </array>
   </map>
</map>
```

Converting a binary media type into XML

This example transforms a piece of text that has been given the (bogus) media type of `x/x` into XML. Because XProc does not recognize this media type, it treats the document as binary. The result of the `p:cast-content-type` step is the document's `base64` encoded contents, wrapped in a `<c:data>` element.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source">
    <p:inline content-type="x/x">Hi there!</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:cast-content-type content-type="text/xml"/>

</p:declare-step>
```

Result document:

```
<c:data xmlns:c="http://www.w3.org/ns/xproc-step"
        content-type="x/x"
        encoding="base64">SGkgdGhlcmUh</c:data>
```

Additional details

- If the value of the `content-type` option and the media type of a document are the same, the document will appear unchanged on the `result` port.
- `p:cast-content-type` preserves all document-properties of the document(s) appearing on its `source` port.

  Exceptions are the `content-type` document-property which is updated accordingly and the `serialization` document-property which is sometimes removed.

Errors raised

Error code	Description
`XC0071` (pg. 217)	It is a dynamic error if the `<p:cast-content-type>` step cannot perform the requested cast.
`XC0072` (pg. 217)	It is a dynamic error if the `<c:data>` contains content is not a valid base64 string.
`XC0073` (pg. 217)	It is a dynamic error if the `<c:data>` element does not have a `@content-type` attribute.
`XC0074` (pg. 217)	It is a dynamic error if the `content-type` is supplied and is not the same as the `@content-type` specified on the `<c:data>` element.
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XD0049` (pg. 220)	It is a dynamic error if the text value is not a well-formed XML document
`XD0057` (pg. 220)	It is a dynamic error if the text document does not conform to the JSON grammar, unless the parameter liberal is true and the processor chooses to accept the deviation.

Error code	Description
XD0058 (pg. 220)	It is a dynamic error if the parameter duplicates is reject and the text document contains a JSON object with duplicate keys.
XD0059 (pg. 220)	It is a dynamic error if the parameter map contains an entry whose key is defined in the specification of fn:parse-json and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter escape with true() is also present.
XD0060 (pg. 220)	It is a dynamic error if the text document can not be converted into the XPath data model
XD0079 (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *type/subtype+ext* " or " *type/subtype* ".

2.7 p:compare

Compares documents for equality.

Summary

```
<p:declare-step type="p:compare">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <input port="alternate" primary="false" content-types="any" sequence="false"/>
  <output port="differences" primary="false" content-types="any" sequence="true"/>
  <option name="fail-if-not-equal" as="xs:boolean" required="false" select="false()"/>
  <option name="method" as="xs:QName?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The p:compare step compares the documents appearing on its source and alternate for equality. It returns a simple XML document containing the boolean result of the comparison.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
source	input	true	any	false	Source document to compare.
result	output	true	application/xml	false	An XML document consisting of a single <c:result> element (the c prefix here is bound to the http://www.w3.org/ns/xproc-step namespace) containing true when the documents compare as equal or false when differences were found. Example: <c:result xmlns:c="http://www.w3.org/ns/xproc-step">true</c:result>
alternate	input	false	any	false	Source document to compare.
differences	output	false	any	true	If the fail-if-not-equal option is false and differences were found, this port *may* submit a summary of the differences. The existence and format of this summary document is implementation-defined and therefore depends on the XProc processor used.

Options:

Name	Type	Req?	Default	Description
`fail-if-not-equal`	`xs:boolean`	`false`	`false`	If this option is `true` and the documents do not compare as equal, the step raises error `XC0019` (pg. 31).
`method`	`xs:QName?`	`false`	`()`	Specifies the comparison method used. If the value of this option is the empty sequence (default) or `deep-equal`, `p:compare` must do the same as the XPath `deep-equal()` (https://www.w3.org/TR/xpath-functions-31/#func-deep-equal) function. Support for any other comparison method is implementation-defined and therefore depends on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the comparison. Keys, values and their meaning depend on the value of the `method` option and the XProc processor used.

Description

The `p:compare` step takes the documents appearing on its `source` and `alternate` ports and tests whether these are equal. Now testing XML documents for equality is not as easy as it sounds: what to do with whitespace, comments, order of attributes, etc. The default behavior of `p:compare` is the same as that of the XPath `deep-equal()` (https://www.w3.org/TR/xpath-functions-31/#func-deep-equal) function. Whether other comparison methods are supported is implementation-defined and therefore dependent on the XProc processor used.

If the `fail-if-not-equal` option is `false` (default), the step emits a simple XML document on its `result` port, saying `true` (equal) or `false` (not equal). If the `fail-if-not-equal` option is `true` and the documents are not equal, error `XC0019` (pg. 31) is raised.

Examples

Comparing two documents

The following document is compared against what we supply on the `alternate` port. In this example the comparison checks out and `p:compare` returns `true`.

```
<texts>
   <text>Hi there!</text>
</texts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:compare>
    <p:with-input port="alternate">
      <texts>
        <text>Hi there!</text>
      </texts>
    </p:with-input>
  </p:compare>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">true</c:result>
```

Additional details

- No document-properties from the documents on the `source` and/or `alternate` ports survive. The resulting document has a `content-type` document-property set to `application/xml` and no `base-uri` document-property.

Errors raised

Error code	Description
`XC0019` (pg. 216)	It is a dynamic error if the documents are not equal according to the specified comparison `method`, and the value of the `fail-if-not-equal` option is `true`.
`XC0076` (pg. 217)	It is a dynamic error if the comparison `method` specified in `<p:compare>` is not supported by the implementation.
`XC0077` (pg. 217)	It is a dynamic error if the media types of the documents supplied are incompatible with the comparison `method`.

2.8 p:compress

Compresses a document.

Summary

```
<p:declare-step type="p:compress">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="format" as="xs:QName" required="false" select="'gzip'"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:compress` step serializes the document appearing on its `source` port and outputs a compressed version of this on its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The document to compress.
`result`	`output`	`true`	`any`	`false`	The resulting compressed document

Options:

Name	Type	Req?	Default	Description
`format`	`xs:QName`	`false`	`gzip`	Specifies the format of the source document. The value `gzip` (default) results in a document compressed using the GZIP (https://datatracker.ietf.org/doc/html/rfc1952) format. Support for any other compression format is implementation-defined and therefore dependent on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the compression. Keys, values and their meaning depend on the value of the `method` option and the XProc processor used.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	Before the document is compressed, it is first serialized (as if written to disk). This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/), controlling this serialization. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence).

Description

The `p:compress` step first serializes the document appearing on its `source`. It then compresses the outcome of this serialization using the format as specified in the `format` option. The result, usually a binary document, appears on its `result` port.

The only compression format that must be supported is GZIP (https://datatracker.ietf.org/doc/html/rfc1952). Support for any other compression format is implementation-defined and therefore dependent on the XProc processor used.

The reverse uncompress operation is supported by the `p:uncompress` (pg. 152) step.

Additional details

- `p:compress` preserves all document-properties of the document(s) appearing on its `source` port.
  Exceptions are the `content-type` document-property which is updated accordingly and the `serialization` document-property which is removed.

Errors raised

Error code	Description
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XC0202` (pg. 219)	It is a dynamic error if the compression format cannot be understood, determined and/or processed.

2.9 p:count

Count the number of documents.

Summary

```
<p:declare-step type="p:count">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="limit" as="xs:integer" required="false" select="0"/>
</p:declare-step>
```

The `p:count` step counts the number of documents appearing on its `source` port and returns an XML document on its `result` port containing that number.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The sequence of documents to count.
`result`	`output`	`true`	`application/xml`	`false`	An XML document consisting of a single `<c:result>` element containing the number of documents on the `source` port, or the limit set by the `limit` option (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). Example: `<c:result xmlns:c="http://www.w3.org/ns/xproc-step">3</c:result>`

Options:

Name	Type	Req?	Default	Description
`limit`	`xs:integer`	`false`	`0`	If the value of this option is greater than 0, the `p:count` will count at most that many documents. See the Limiting the count (pg. 34) example. Since `p:count` will stop processing documents on its `source` when this limit is reached, this provides an efficient mechanism to discover if a sequence consists of more than X documents.

Description

The `p:count` step simply counts the number of documents appearing on its `source` port. It emits a very simple document on its `result` port containing this number, wrapped in a `<c:result>` element (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). For example: `<c:result>3</c:result>`.

Using this step you can find out how many documents are flowing through your pipeline and make decision based on that number. For instance: stop processing if there are too many.

There is another way of doing this, see the Alternative to p:count (pg. 34) example. This is probably easier, because the required count is directly in a variable and the flow of documents in the pipeline is not interrupted. Which method (`p:count` or a count variable) is faster is hard to say and probably dependent on the XProc processor used. If this matters to you, you'll have to do some experiments.

Examples

Basic usage

The following pipeline produces a sequence of 3 documents and counts these:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
  </p:input>
  <p:output port="result"/>

  <p:count/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">3</c:result>
```

Using this count to make a decision could be done as follows:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
  </p:input>
  <p:output port="result"/>

  <p:count/>

  <p:choose>
    <p:when test="number(.) eq 3">
      <p:identity>
        <p:with-input>
          <count-is-exactly-3/>
        </p:with-input>
      </p:identity>
    </p:when>
    <p:otherwise>
      <p:identity>
        <p:with-input>
          <count-is-not-3/>
        </p:with-input>
      </p:identity>
    </p:otherwise>
  </p:choose>


</p:declare-step>
```

Result document:

```
<count-is-exactly-3/>
```

Limiting the count

The following pipeline produces a sequence of 3 documents and counts these with a limit of 1:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
  </p:input>
  <p:output port="result"/>

  <p:count limit="1"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">1</c:result>
```

Alternative to p:count

You don't need `p:count` to count documents. An alternative is to declare a variable that gets its value by counting the size of the collection of documents flowing through it:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
    <p:document href="in1.xml"/>
  </p:input>
  <p:output port="result"/>

  <p:variable name="count" select="count(collection())" collection="true"/>

  <p:identity>
    <p:with-input>
      <document-count>{$count}</document-count>
    </p:with-input>
  </p:identity>

</p:declare-step>
```

Result document:

```
<document-count>3</document-count>
```

The `p:identity` after the variable declaration is only there for demonstration purposes: to show the value of the count. In a real pipeline you would probably follow it up with a `<p:if>` or `<p:choose>`/`<p:when>`/`<p:otherwise>` and make some decision based on the document count.

Additional details

- No document-properties from the documents on the `source` port survive. The resulting document has a `content-type` document-property set to `application/xml` and no `base-uri` document-property.

2.10 p:css-formatter

Renders a document using CSS formating.

Summary

```
<p:declare-step type="p:css-formatter">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <input port="stylesheet" primary="false" content-types="text" sequence="true">
    <p:empty/>
  <input/>
  <option name="content-type" as="xs:string?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:css-formatter` step renders an XML or HTML document using CSS formatting (https://www.w3.org/TR/css-2018/), usually into PDF. The CSS stylesheets for this must be present on the `stylesheet` port. The resulting rendition appears, as a binary document, on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The XML or HTML document to render.
`result`	`output`	`true`	`any`	`false`	The resulting rendition.
`stylesheet`	`input`	`false`	`text`	`true`	The CSS stylesheets to use for the rendering.

Options:

Name	Type	Req?	Default	Description
`content-type`	`xs:string?`	`false`	`()`	The content-type (media type) of the rendition that appears on the `result` port. The default value is `application/pdf`. Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor and renderer used. This option may include media type parameters as well (for instance `application/pdf; charset=UTF-8`).
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the rendering. The XProc specification does not define any parameters for this option. A specific XProc processor (or renderer used) might define its own.

Description

The `p:css-formatter` step allows you to transform XML or HTML into some kind of rendition, usually PDF, by applying CSS formatting (https://www.w3.org/TR/css-2018/). Most likely, CSS Paged Media (https://www.w3.org/TR/css-page-3/) will be used.

In most cases, `p:css-formatter` relies on an external CSS formatter. You'll probably have to do some XProc processor dependent configuration before this step will work. Please consult the XProc processor documentation about this.

Examples

Basic usage

The following pipeline transforms some XML document into PDF using `p:css-formatter`, and stores it as `result.pdf`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="result-uri@store-pdf"/>

  <p:css-formatter content-type="application/pdf">
    <p:with-input port="stylesheet" href="my-paged-media-stylesheet.css"/>
  </p:css-formatter>
  <p:store href="result.pdf" name="store-pdf"/>

</p:declare-step>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0166` (pg. 219)	It is a dynamic error if the requested document cannot be produced.
`XC0204` (pg. 220)	It is a dynamic error if the requested content-type is not supported.
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.11 p:delete

Delete nodes in documents.

Summary

```
<p:declare-step type="p:delete">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <option name="match" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:delete` step deletes nodes, specified by an XSLT selection pattern, from the document appearing on its `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to delete the nodes from.
`result`	`output`	`true`	`text` `xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Description
`match`	`xs:string` (XSLT selection pattern)	`true`	The XSLT match pattern for the nodes to delete, as a string.

Description

The `p:delete` step takes the XSLT match pattern in the `match` option and holds this against the document appearing on its `source` port. Any matching nodes are deleted (including child nodes, if any). The resulting document appears on the `result` port.

Note that deleting an element means that the entire element, including child elements, gets deleted. If you just want to delete the element but keep its child elements, you need `p:unwrap` (pg. 154).

Examples

Basic usage

The following example deletes all `<text>` elements with an attribute `type="normal"` from the source document.

Source document:

```
<texts>
   <text>Hello there!</text>
   <text>This is funny…</text>
   <text type="normal">And that's normal.</text>
   <text type="normal">Very normal…</text>
</texts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:delete match="text[@type eq 'normal']"/>

</p:declare-step>
```

Result document:

```
<texts>
   <text>Hello there!</text>
   <text>This is funny…</text>
</texts>
```

Additional details

- `p:delete` preserves all document-properties of the document(s) appearing on its `source` port.
- This step cannot remove namespaces, if you try this `XC0023` (pg. 37) is raised. For removing namespaces use `p:namespace-delete` (pg. 101).
- Deleting an `xml:base` attribute does *not* change the base URI of the element on which it occurred.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.
`XC0062` (pg. 216)	It is a dynamic error if the `match` option matches a namespace node.

2.12 p:directory-list

List the contents of a directory.

Summary

```
<p:declare-step type="p:directory-list">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="path" as="xs:anyURI" required="true"/>
  <option name="detailed" as="xs:boolean" required="false" select="false()"/>
  <option name="exclude-filter" as="xs:string*" required="false" select="()"/>
  <option name="include-filter" as="xs:string*" required="false" select="()"/>
  <option name="max-depth" as="xs:string?" required="false" select="'1'"/>
  <option name="override-content-types" as="array(array(xs:string))?" required="false" select="()"/>
</p:declare-step>
```

The `p:directory-list` step produces an XML document that contains an overview of the contents of a specified directory.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	The resulting XML document that describes the contents of the directory. See "The result document" on page 39.

Options:

Name	Type	Req?	Default	Description
`path`	`xs:anyURI`	`true`		The path of the directory to describe the contents of.
`detailed`	`xs:boolean`	`false`	`false`	Whether detailed information about the directory and its contents is returned. See TBD
`exclude-filter`	`xs:string*`	`false`	`()`	A sequence of XPath regular expression that specifies which directories/files are excluded. See "Including and excluding files and directories" on page 39. See also the Including and excluding files (pg. 42) example.

Name	Type	Req?	Default	Description
`include-filter`	`xs:string*`	`false`	`()`	A sequence of XPath regular expression that specifies which directories/files are included. See "Including and excluding files and directories" on page 39. See also the Including and excluding files (pg. 42) example.
`max-depth`	`xs:string?`	`false`	`1`	How deep (how many levels of subdirectories) the directory is described. Its value must be a string that can be cast to either a (non-negative) integer or the word `unbounded`: • A value of `0` means that only information about the given directory is returned. • A value of `1` (default) returns information about the direct contents of the given directory. • A numerical value greater than `1` returns information up to that level of subdirectories. • A value `unbounded` returns information about all subdirectories. See also the Changing the depth of the directory lisiting (pg. 41) example.
`override-content-types`	`array(array (xs:string))?`	`false`	`()`	Use this to override the content-type determination of files. Determining the content-type of a file happens when you ask for detailed information (the `detailed` option is set to `true`). This works just like the mechanism for the `override-content-types` option of `p:archive-manifest` (pg. 18), except that the regular expression matching is done against the paths as used for the matching of the `include-filter` and `exclude-filter` options. For more information see "Including and excluding files and directories" on page 39.

Description

The `p:directory-list` step provides you with an overview of the contents of a directory, similar to a Windows `dir` or a Unix/Linux/macOS `ls` command. This often comes in handy, for instance when you need to perform some operation on *all* files in a directory (or a directory tree). The examples Handling all files in a directory (A) (pg. 43) and Handling all files in a directory (B) (pg. 44) show how you could do this.

The `p:directory-list` step takes a directory path as its main input in the `path` option. The `result` port emits a document (see "The result document" on page 39) that describes this directory by listing its contents (files and subdirectories). What happens exactly depends on the settings of the other options. The step has no input port(s).

The directory to describe, as specified in the `path` option, *must* exist. Otherwise, error `XC0017` (pg. 45) is raised.

Including and excluding files and directories

The `include-filter` and `exclude-filter` determine which files and directories are included/excluded in the result. Both options are a sequence of (zero or more) XPath regular expression strings.

- If the `include-filter` is not specified (or the empty sequence), *all* files/directories are included.

  Otherwise, every regular expression string in the option value is matched against the *relative* file/directory paths (relative to the path that was given in the `path` option). A match means the file/directory is included.
- If the `exclude-filter` is not specified (or the empty sequence), *no* files/directories are excluded.

  Otherwise, every regular expression string in the option value is matched against the *relative* file/directory paths (relative to the path that was given in the `path` option). A match means the file/directory is excluded.
- A file/directory is part of the result if it is included and not excluded.

Matching the regular expressions behaves like applying the XPath `matches() (https://www.w3.org/TR/xpath-functions-31/#func-matches)` function (like in `matches($relative-path, $regular-expression)`).

The result document

The root element of the resulting XML document is `<c:directory>` (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace):

```
<c:directory name = xs:string
             xml:base = xs:anyURI
             hidden? = xs:boolean
             last-modified? = xs:dateTime
             readable? = xs:boolean
             size? = xs:integer
             writable? = xs:boolean >
  ( <c:file> |
    <c:directory> |
    <c:other>  )*
</c:directory>
```

Attribute	#	Type	Description
`name`	1	`xs:string`	The name of the directory (without a path in front).
`xml:base`	1	`xs:anyURI`	The URI of the directory, always ending with a slash. • For the root `<c:directory>` element this will be the absolute path of the directory described. • For any nested `<c:directory>` elements, this will be the name of the directory.
`hidden`	?	`xs:boolean`	Whether this directory is hidden for the current user. See below.
`last-modified`	?	`xs:dateTime`	The date and time this directory was last modified. See below.
`readable`	?	`xs:boolean`	Whether this directory is readable for the current user. See below.
`size`	?	`xs:integer`	The size of the directory entry (in bytes). See below.
`writable`	?	`xs:boolean`	Whether this directory is readable for the current user. See below.

Child element	#	Description
`c:file`	*	An file in the given directory
`c:directory`	*	A subdirectory in the given directory
`c:other`	*	Anything in the given directory that is “special”. What is considered special is implementation defined and therefore depends on the XProc processor used.

Every file in a directory is described using a `<c:file>` element:

```
<c:file name = xs:string
        xml:base = xs:anyURI
        content-type? = xs:string
        hidden? = xs:boolean
        last-modified? = xs:dateTime
        readable? = xs:boolean
        size? = xs:integer
        writable? = xs:boolean />
```

Attribute	#	Type	Description
`name`	1	`xs:string`	The name of the file (without a path in front).
`xml:base`	1	`xs:anyURI`	The name of the file (identical to the `name` attribute).
`content-type`	?	`xs:string`	The content-type (MIME type) of this file. If this cannot be determined, its value is `application/octet-stream`. See below.
`hidden`	?	`xs:boolean`	Whether this file is hidden for the current user. See below.
`last-modified`	?	`xs:dateTime`	The date and time this file was last modified. See below.
`readable`	?	`xs:boolean`	Whether this file is readable for the current user. See below.
`size`	?	`xs:integer`	The size of the file entry (in bytes). See below.
`writable`	?	`xs:boolean`	Whether this file is readable for the current user. See below.

Anything else in a directory is described using the `<c:other>` element. This looks just like the `<c:file>` element, but without a `content-type` attribute.

About the optional attributes on the result elements:

- If the `detailed` option is `false` (default), only the `name` and `xml:base` attributes will be there.
- If the `detailed` option is `true`, the other, optional, attributes will be present also.

What the values of the various attributes actually mean is implementation defined and therefore depends on the XProc processor used. For most attributes there will be no surprises, but what, for instance, is the size of a directory? It may take some experiments to get things right.

Examples

Basic usage

Assume we have a disk layout that looks like this:

```
-- data -- + -- x1.txt
           |
           + -- x1.xml
           |
           + -- sub1/ -- + -- sub1-x1.xml
                         |
                         + -- sub2/ -- + -- sub2.tmp
                                       |
                                       + -- sub2-x1.txt
```

For the examples to come we assume this `data` directory is in the same location as our pipeline. Simply asking for the directory listing, using the default values for the options of `p:directory-list`, is as follows:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data">
   <c:directory xml:base="sub1/" name="sub1"/>
   <c:file xml:base="x1.txt" name="x1.txt"/>
   <c:file xml:base="x1.xml" name="x1.xml"/>
</c:directory>
```

When we ask for details, the following happens:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" detailed="true"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xmlns:mox="http://www.xml-project.com/morganaxproc"
             xml:base="file:/…/…/data/"
             name="data"
             readable="true"
             writable="true"
             mox:executable="true"
             hidden="false"
             last-modified="2025-06-03T11:42:32.4Z"
             size="0">
   <c:directory xml:base="sub1/"
                name="sub1"
                readable="true"
                writable="true"
                mox:executable="true"
                hidden="false"
                last-modified="2025-06-03T11:42:32.38Z"
                size="0"/>
   <c:file xml:base="x1.txt"
           name="x1.txt"
           content-type="text/plain"
           readable="true"
           writable="true"
           mox:executable="true"
           hidden="false"
           last-modified="2024-12-27T11:30:00.96Z"
           size="0"/>
   <c:file xml:base="x1.xml"
           name="x1.xml"
           content-type="application/xml"
           readable="true"
           writable="true"
           mox:executable="true"
           hidden="false"
           last-modified="2025-02-05T12:05:59.74Z"
           size="83"/>
</c:directory>
```

Changing the depth of the directory lisiting

The following examples work on the same directory structure as described in Basic usage (pg. 40). Asking for a directory description with `max-depth` option set to `0` just gives us the main directory itself:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" max-depth="0"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data"/>
```

And getting the full directory structure is as follows:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" max-depth="unbounded"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data">
   <c:directory xml:base="sub1/" name="sub1">
      <c:file xml:base="sub1-x1.xml" name="sub1-x1.xml"/>
      <c:directory xml:base="sub2/" name="sub2">
         <c:file xml:base="sub2-x1.txt" name="sub2-x1.txt"/>
         <c:file xml:base="sub2.tmp" name="sub2.tmp"/>
      </c:directory>
   </c:directory>
   <c:file xml:base="x1.txt" name="x1.txt"/>
   <c:file xml:base="x1.xml" name="x1.xml"/>
</c:directory>
```

Including and excluding files

The following examples work on the same directory structure as described in Basic usage (pg. 40). Assume we only need the text files in the directory tree: all files ending with `.txt`. A regular expression that matches this is `\.txt$`, so we have to pass this as the value of the `include-filter` option:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" include-filter="\.txt$" max-depth="unbounded"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data">
   <c:directory xml:base="sub1/" name="sub1">
      <c:directory xml:base="sub2/" name="sub2">
         <c:file xml:base="sub2-x1.txt" name="sub2-x1.txt"/>
      </c:directory>
   </c:directory>
   <c:file xml:base="x1.txt" name="x1.txt"/>
</c:directory>
```

Assume that we know that all files that start with an `x` are not interesting. We can exclude these by passing the regular expression `^x` as the value of the `exclude-filter` option:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" include-filter="\.txt$" exclude-filter="^x" max-depth="unbounded"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data">
   <c:directory xml:base="sub1/" name="sub1">
      <c:directory xml:base="sub2/" name="sub2">
         <c:file xml:base="sub2-x1.txt" name="sub2-x1.txt"/>
      </c:directory>
   </c:directory>
</c:directory>
```

Finally, assume we both need the XML and text files in the directory tree, but not anything else. For this we could do two things:

- Create a regular expression that incorporates both, and pass it as an `include-filter` attribute on the `<p:directory-list>` element, just like we did in the examples above: `<p:directory list path="data" include-filter="\.(xml|txt)$" max-depth="unbounded"/>`
- Or we could pass a regular expression for each file type. If we do it this way we can no longer pass the `include-filter` option as an attribute. We have to use a `<p:with-option>` child element:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" max-depth="unbounded">
    <p:with-option name="include-filter" select="('\.xml$', '\.txt$')"/>
  </p:directory-list>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data">
   <c:directory xml:base="sub1/" name="sub1">
      <c:file xml:base="sub1-x1.xml" name="sub1-x1.xml"/>
      <c:directory xml:base="sub2/" name="sub2">
         <c:file xml:base="sub2-x1.txt" name="sub2-x1.txt"/>
      </c:directory>
   </c:directory>
   <c:file xml:base="x1.txt" name="x1.txt"/>
   <c:file xml:base="x1.xml" name="x1.xml"/>
</c:directory>
```

Handling all files in a directory (A)

Again, the following examples work on the same directory structure as described in Basic usage (pg. 40). Assume we need to do something with all XML documents in the `data` directory. Using `p:directory-list` we can easily get the names of these files. However, to process them we will need to load them, and for that its handy if we have their full absolute URIs.

There are several ways to do this. One, using the `p:make-absolute-uris` (pg. 97) step, is shown in this example. Another one is shown in the Handling all files in a directory (B) (pg. 44) example below.

Using the `p:make-absolute-uris` (pg. 97) step we can change the `name` attributes into full URIs:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" include-filter="\.xml$" max-depth="unbounded"/>
  <p:make-absolute-uris match="@name"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="file:/…/…/data/data">
   <c:directory xml:base="sub1/" name="file:/…/…/data/sub1/sub1">
      <c:file xml:base="sub1-x1.xml" name="file:/…/…/data/sub1/sub1-x1.xml"/>
   </c:directory>
   <c:file xml:base="x1.xml" name="file:/…/…/data/x1.xml"/>
</c:directory>
```

We can now use this result to process all the XML documents. The following pipeline simply loads them (using `p:load` (pg. 94)), and wraps all contents (using `p:wrap-sequence` (pg. 187)) in an `<all-xml-documents>` element:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" include-filter="\.xml$" max-depth="unbounded"/>
  <p:make-absolute-uris match="@name"/>

  <p:for-each>
    <p:with-input select="//c:file"/>
    <p:load href="{/*/@name}"/>
  </p:for-each>
  <p:wrap-sequence wrapper="all-xml-documents"/>

</p:declare-step>
```

Result document:

```
<all-xml-documents>
   <data>This is document sub1/sub1-x1.xml</data>
   <data>This is document data/x1.xml</data>
</all-xml-documents>
```

Handling all files in a directory (B)

Another way to approach the problem in example Handling all files in a directory (A) (pg. 43) is using the XPath `base-uri()` (https://www.w3.org/TR/xpath-functions-31/#func-base-uri) function on the `<c:file>` elements. This works because all `<c:directory>` and `<c:file>` elements in the result of `p:directory-list` have an `xml:base` attribute. These together make the base URIs of the `<c:file>` elements the URIs of the documents referenced.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:directory-list path="data" include-filter="\.xml$" max-depth="unbounded"/>
  <p:make-absolute-uris match="@name"/>

  <p:for-each>
    <p:with-input select="//c:file"/>
    <p:load href="{base-uri(/*)}"/>
  </p:for-each>
  <p:wrap-sequence wrapper="all-xml-documents"/>

</p:declare-step>
```

Result document:

```
<all-xml-documents>
   <data>This is document sub1/sub1-x1.xml</data>
   <data>This is document data/x1.xml</data>
</all-xml-documents>
```

Additional details

- Only the `base-uri` property will be set. Its value will be the absolute URI of the directory described.
- A relative value for the `path` option is resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:directory-list` is stored).
- If some entry (file or directory) is included in the result, all directories leading up to this entry are always included, even if they're excluded because of the `include-filter` and `exclude-filter` option settings. This assures that the hierarchy of the result always matches the hierarchy of the filesystem.

- Working on “normal” files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a “directory” and “file” may vary.
- An XProc processor may add additional, implementation-defined, attributes to the various result elements as described in “The result document” on page 39. These attributes will always be in some, XProc processor dependent, namespace.

Errors raised

Error code	Description
`XC0012` (pg. 216)	It is a dynamic error if the contents of the directory path are not available to the step due to access restrictions in the environment in which the pipeline is run.
`XC0017` (pg. 216)	It is a dynamic error if the absolute path does not identify a directory.
`XC0090` (pg. 217)	It is a dynamic error if an implementation does not support directory listing for a specified scheme.
`XC0147` (pg. 219)	It is a dynamic error if the specified value is not a valid XPath regular expression.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.13 p:encode

Encodes a document.

Summary

```
<p:declare-step type="p:encode">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="application/xml" sequence="true"/>
  <option name="encoding" as="xs:string" required="false" select="()"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:encode` step encodes the document appearing on its `source` port, for example with `base64` encoding. The encoded version is wrapped in a `<c:data>` element and appears on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The source document to encode: • If this is a binary document, its content is encoded directly. • Any other kind of document is serialized first (as if written to disk, see also the `serialization` option). The serialized version is encoded.
`result`	`output`	`true`	`application/xml`	`true`	A `<c:data>` element (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace), containing the encoded version of the source document. See here (https://xprocref.org/3.1/p.cast-content-type.html#c-data) for a more detailed description of the `<c:data>` element.

Options:

Name	Type	Req?	Default	Description
`encoding`	`xs:string`	`false`	`()`	The encoding the step must produce. The only standard value currently supported is `base64` (default). Whether any other encodings are supported and what their names (value for this option) are is implementation-defined and therefore dependent on the XProc processor used.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/) for serializing the document on the `source` port, before encoding takes place. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence). Example: `serialization="map{'indent': false()}"`. See also the Effect of serialization (pg. 47) example.

Description

The `p:encode` can be used to encode a document. The only standard encoding currently supported is `base64`. The encoded version of the source document is wrapped in a `<c:data>` element and appears on the `result` port. A more detailed description of the `<c:data>` element can be found here (https://xprocref.org/3.1/p.cast-content-type.html#c-data).

There is no `p:decode` step. Decoding (of `<c:data>` elements) is performed by the `p:cast-content-type` (pg. 22) step.

Examples

Basic usage

Assume we have a simple input text document that looks like this:

```
Hi there!
```

Feeding this to `p:encode` results in the following:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:encode/>

</p:declare-step>
```

Result document:

```
<c:data xmlns:c="http://www.w3.org/ns/xproc-step"
        content-type="text/plain"
        encoding="base64"
        charset="UTF-8">SGkgdGhlcmUh</c:data>
```

Effect of serialization

When encoding an XML document, this is serialized first (as if written to disk). The result of the serialization therefore has an effect on the outcome, as shown in the two examples below.

Source document:

```
<text>
   <para>Hello XProc fans!</para>
</text>
```

Encoding *with* indenting:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:encode serialization="map{'indent': true()}"/>

</p:declare-step>
```

Result document:

```
<c:data xmlns:c="http://www.w3.org/ns/xproc-step"
        content-type="application/xml"
        encoding="base64"
        charset="UTF-8">PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4KPHRleHQ
+CiAgIDxwYXJhPkhlbGxvIFhQcm9jIGZhbnMhPC9wYXJhPgo8L3RleHQ+</c:data>
```

Encoding *without* indenting (using the same input document as in the example above):

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:encode serialization="map{'indent': false()}"/>

</p:declare-step>
```

Result document:

```
<c:data xmlns:c="http://www.w3.org/ns/xproc-step"
        content-type="application/xml"
        encoding="base64"
        charset="UTF-8">PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz48dGV4dD4KICA8cGFyYT5IZWxsbyBYUHJvYyBmYW5zITwvcGFyYT4KPC90ZXh0Pg==
c:data>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

2.14 p:error

Raises an error.

Summary

```
<p:declare-step type="p:error">
  <input port="source" primary="true" content-types="text xml" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <option name="code" as="xs:QName" required="true"/>
</p:declare-step>
```

The `p:error` step raises a (dynamic) error, using the value of the `code` option as the error code. The document(s) on its `source` port become the error message(s).

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text` `xml`	`true`	The contents/message of the error raised.
`result`	`output`	`true`	`any`	`true`	This port is just there for the convenience of pipeline authors. Nothing will ever appear on this port (since `p:error` stops execution of the pipeline by raising an error).

Options:

Name	Type	Req?	Description
`code`	`xs:QName`	`true`	The code for the error raised.

Description

the `p:error` step raises a dynamic error, breaking the pipeline's document flow.

An error has a code, which must be provided using the `code` option. An error code is a QName (a name with an optional namespace part). This code is shown in the resulting error message. You can also use this code for catching this specific error in a `p:try/p:catch` construction. Using a namespace in an error code raised by `p:error` is recommended because it then clearly distinguishes itself from errors raised by XProc itself.

The text or XML document(s) on the step's `source` port become the error message(s) accompanying the error. They will return as the contents of the `c:errors/c:error` element(s) in the error report document (https://spec.xproc.org/3.1/xproc/#err-vocab) produced by the error.

The `p:error` step also has a (primary) output port, but that is just for the convenience of the pipeline author: nothing will ever appear on it (since the flow is broken by the generated error). It makes it easier to insert a `p:error` in situations where a primary output port is required, for instance inside a `p:if` that tests whether an error must be raised.

Examples

Basic usage

The following example raises an error using `p:error`. Please notice that we use a namespace for the error code (which is recommended but not required).

Source document:

```
<result status="bad"/>
```

The example pipeline checks the result status. Watch out: the `p:try/p:catch` construction that surrounds `p:error` is there for *example purposes only* (don't try this at home)! It takes care of showing the resulting error report document as the step's result.

```
<p:declare-step xmlns:my="#my-application" xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:if test="/*/@status ne 'good'">
    <p:try>

      <p:error code="my:error">
        <p:with-input>
          <message>The status is not good but {/*/@status}</message>
        </p:with-input>
      </p:error>

      <p:catch name="error-catch">
        <p:identity>
          <p:with-input pipe="error@error-catch"/>
        </p:identity>
      </p:catch>

    </p:try>
  </p:if>

</p:declare-step>
```

The resulting error report document:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:error code="err:XD0052"
            name="!1.1.1.1.1-source-0"
            type="!1.1.1.1.1-source-0"
            href="file:/…/…/error-01.xpl"
            line="11"
            column="32">
      <message>Unable to build inline document with TVT: Unable to add free standing attribute here.</
message>
   </c:error>
</c:errors>
```

Notice that in the example above the error message is inside a `<message>` element. Usually however, error messages are just text, strings. This can be accomplished by providing the error message on the `p:error source` port as a text document:

```
<p:declare-step xmlns:my="#my-application" xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:if test="/*/@status ne 'good'">
    <p:try>

      <p:error code="my:error">
        <p:with-input>
          <p:inline content-type="text/plain">The status is not good but {/*/@status}</p:inline>
        </p:with-input>
      </p:error>

      <p:catch name="error-catch">
        <p:identity>
          <p:with-input pipe="error@error-catch"/>
        </p:identity>
      </p:catch>

    </p:try>
  </p:if>

</p:declare-step>
```

The resulting error report document:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:error code="err:XD0084"
            name="!1.1.1.1.1-source-0"
            type="!1.1.1.1.1-source-0"
            href="file:/…/…/error-02.xpl"
            line="11"
            column="32">
      <message>Unable to build inline text document with TVT. Free standing attributes not allowed.</
message>
   </c:error>
</c:errors>
```

Additional details

- If more than one document appears on the `source` port of `p:error`, all source documents become children of a single `<p:error>` element.

2.15 p:file-copy

Copies a file or directory.

Summary

```
<p:declare-step type="p:file-copy">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="target" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="overwrite" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:file-copy` step copies a file or a directory to a given target.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the target (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the source file or directory to copy from.
`target`	`xs:anyURI`	`true`		The URI of the target file or directory to copy to.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port. If you're copying a directory, please note that an error may mean that a partial copy has already been made.
`overwrite`	`xs:boolean`	`false`	`true`	Determines what happens if `p:file-copy` needs to overwrite an existing file: • If this option is `true` (default), the existing file is overwritten. • If this option is `false`, no existing file will be changed.

Description

The `p:file-copy` step copies a file or a directory, as specified in the `href` option, to the target specified in the `target` option. Any non-existent directory in the `target` option value will be created. The `result` port emits a small XML document with only a `<c:result>` element containing the absolute URI of the target (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

If the `target` option specifies an existing directory (existing on disk), the step attempts to copy the source file or directory *into* that target directory, preserving the name of the source. This means that you cannot use `p:file-copy` to copy a directory to another location under a different name. See the Copying a directory under a different name (pg. 51) example on how to achieve this.

Examples

Basic usage

The following example copies a file `data/x1.xml` to `build/x1-copied`:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-copy href="data/x1.xml" target="build/x1-copied.xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/x1-copied.xml</c:result>
```

Copying a directory under a different name

If you're copying a directory, the `p:file-copy` step always copies this *into* the designated target directory. This means that you cannot copy a directory to another location under a different name. In order to achieve this, you subsequently must rename the copied result using `p:file-move` (pg. 60):

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-copy href="data/" target="build/"/>
  <p:file-move href="build/data/" target="build/data-renamed/"/>

</p:declare-step>
```

Result document (of the `p:file-move` (pg. 60)):

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/data-renamed/</c:result>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` and `target` options are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-copy` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0050` (pg. 216)	It is a dynamic error the file or directory cannot be copied to the specified location.
`XC0144` (pg. 219)	It is a dynamic error if an implementation does not support `<p:file-copy>` for a specified scheme.
`XC0145` (pg. 219)	It is a dynamic error if `<p:file-copy>` is not available to the step due to access restrictions in the environment in which the pipeline is run.
`XC0157` (pg. 219)	It is a dynamic error if the `href` option names a directory, but the `target` option names a file.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.16 p:file-create-tempfile

Creates a temporary file.

Summary

```
<p:declare-step type="p:file-create-tempfile">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="delete-on-exit" as="xs:boolean" required="false" select="false()"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="href" as="xs:anyURI?" required="false" select="()"/>
  <option name="prefix" as="xs:string?" required="false" select="()"/>
  <option name="suffix" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:file-create-tempfile` step creates a temporary file.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the created temporary file (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`delete-on-exit`	`xs:boolean`	`false`	`false`	If set to `true`, an attempt will be made to automatically delete the temporary file when the processor terminates the pipeline. No error will be raised if this is unsuccessful.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port.
`href`	`xs:anyURI?`	`false`	`()`	The URI of the (existing) directory where the temporary file is created. If not specified, location of the temporary file is implementation-defined and therefore depends on the XProc processor used. Usually this will be the operating system's default location for temporary files.
`prefix`	`xs:string?`	`false`	`()`	The prefix string for the name of the temporary file to create. Specifying a prefix is useful if your want the temporary files created by your pipeline to be distinguishable from other temporary files in the same directory.
`suffix`	`xs:string?`	`false`	`()`	The suffix string for the name of the temporary file to create. Setting the `suffix` option must be used if you want your temporary file to have a specific extension, like `.xml`. See also the Specifying the temporary file's extension (pg. 53) example.

Description

XProc is designed to process documents without having to continuously store and load these documents to/from disk. But sometimes you do need documents to be stored on disk, for instance when some additional process demands this. You can of course solve this by using `p:store` (pg. 132) with a fixed (or generated) filename. However, if the filename is not important, or there is no obvious location for these files, or if you do't want these documents to survive pipeline processing, `p:file-create-tempfile` comes to the rescue.

The `p:file-create-tempfile` step creates a temporary file:

- It is guaranteed that this file didn't exist beforehand.
- The directory where the will be created can be specified using the `href` option (if you don't, some logical, system dependent, location for temporary files will be used).
- The main part of the filename will be generated by the step. However, you can specify a filename suffix (in the `suffix` option) and prefix (in the `prefix` option). This is especially useful if you want your temporary file to have a specific extension. See also the Specifying the temporary file's extension (pg. 53) example.
- If you want the resulting temporary file to get deleted after the pipeline finishes, set the `delete-on-exit` option to true. There is however no guarantee that this deletion succeeds (and you will not get notified if it does't).

The `result` port emits a small XML document with only a `<c:result>` element, containing the absolute URI of the temporary file (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Examples

Basic usage

The following example creates a temporary file in the (existing) `build/` subdirectory (and deletes this again when the pipeline has come to an end):

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-create-tempfile href="build/" delete-on-exit="true"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/533577476312691988.tmp</c:result>
```

Specifying the temporary file's extension

The following example creates a temporary file with a specific `.xml` extension in the (existing) `build/` subdirectory. In this case we don't specify that the file must get deleted after the pipeline finishes. Because of the explicit `.xml` extension, it will be easy to open and inspect the file using your XML editor, if needed.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-create-tempfile href="build/" suffix=".xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/4097921365549781430.xml</c:result>
```

Using a temporary file

Of course, just creating a temporary file is not very useful. You'll want to write some data to it. The following example shows you how to do this: the document appearing on the `source` port is written to the temporary file. This is not very useful in itself but it shows how you can "catch" the created temporary file URI and use this in a subsequent step.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0" name="example-pipeline">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:file-create-tempfile href="build/" suffix=".xml"/>

  <p:store href="{string(.)}">
    <p:with-input pipe="source@example-pipeline"/>
  </p:store>

</p:declare-step>
```

You might also consider catching the created temporary file URI in a variable and use it from there. For this, add the next line directly after the `p:file-create-tempfile` invocation:

```
<p:variable name="href-tempfile-uri" select="string(.)"/>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` option are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-create-tempfile` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0116` (pg. 218)	It is a dynamic error if the temporary file could not be created.
`XC0138` (pg. 219)	It is a dynamic error if an implementation does not support `<p:file-create-tempfile>` for a specified scheme.
`XC0139` (pg. 219)	It is a dynamic error if `<p:file-create-tempfile>` cannot be completed due to access restrictions in the environment in which the pipeline is run.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.17 p:file-delete

Deletes a file or directory.

Summary

```
<p:declare-step type="p:file-delete">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="recursive" as="xs:boolean" required="false" select="false()"/>
</p:declare-step>
```

The `p:file-delete` step deletes a file or directory.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the file or directory to delete (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the file or directory to delete.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port. If you're deleting a directory with the `recursive` option set to `true`, please note that an error may mean that a partial delete has already been made.
`recursive`	`xs:boolean`	`false`	`false`	When deleting a directory, setting this option to `true` means that the contents of the directory (any contained files and/or sub-directories) will also be deleted. When this option is set to `false` (default), deleting a directory is possible only when the directory is empty.

Description

The `p:file-delete` step deletes a file or directory specified in the `href` option. The `result` port emits a small XML document with only a `<c:result>` element, containing the absolute URI of the deleted file or directory (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Be aware that whether or not the file or directory exists, the step will *always* return a `<c:result>` element. It makes no distinction between "deleted" and "didn't exist in the first place". Some XProc processors add additional attributes to the `<c:result>` element (in a separate, processor dependent, namespace) to tell you what really happened.

If you're deleting a directory, you'll probably want the `recursive` option set to `true`. This ensures that all containing files an/or subdirectories will also be deleted (recursively). Deleting a directory with `recursive` set to `false` (default) will succeed only when the directory is empty.

Examples

Basic usage

The following example deletes an (existing) file `data/x.xml`:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-delete href="data/x.xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/data/x.xml</c:result>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` option are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-delete` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0113` (pg. 218)	It is a dynamic error if an attempt is made to delete a non-empty directory and the `recursive` option was set to `false`.
`XC0142` (pg. 219)	It is a dynamic error if an implementation does not support `<p:file-delete>` for a specified scheme.
`XC0143` (pg. 219)	It is a dynamic error if `<p:file-delete>` is not available to the step due to access restrictions in the environment in which the pipeline is run.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.18 p:file-info

Returns information about a file or directory.

Summary

```
<p:declare-step type="p:file-info">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="override-content-types" as="array(array(xs:string))?" required="false" select="()"/>
</p:declare-step>
```

The `p:file-info` step returns information about a file or directory (or other file system object).

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A small XML document, describing the file or directory referenced by the `href` option. See "The result document" on page 57.

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the file or directory to describe.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port.
`override-content-types`	`array(array (xs:string))?`	`false`	`()`	Use this to override the content-type determination of files. This works just like the mechanism for the `override-content-types` option of `p:archive-manifest` (pg. 18), except that the regular expression matching is done against the absolute URI of the file.

Description

The `p:file-info` step returns information about a file or directory (or other file system object) as a small XML document on its `result` port. What will be returned is dependent on the type of file system object, see "The result document" on page 57.

The result document

The result document describing a file or directory consists of either a `<c:file>` or `<c:directory>` element (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). These elements and their attributes are the same as returned by `p:directory-list` (pg. 37) for such a file system object (with the `detailed` option set to `true`).

If the `href` option references any other system object (for instance, on Unix, a device), the result is implementation-defined and therefore depends on the XProc processor used.

Examples

Basic usage

The following example returns information about the `data/` directory

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-info href="data/"/>

</p:declare-step>
```

Result document:

```
<c:directory xmlns:c="http://www.w3.org/ns/xproc-step"
             xml:base="file:/…/…/data/"
             name="data"
             readable="true"
             writable="true"
             hidden="false"
             last-modified="2025-06-03T11:42:32.57Z"
             size="0"/>
```

Notice that the result has an `xml:base` attribute with the absolute URI of the object described. This is attribute is not mandatory, but you can very probably rely on it being there.

Describing a file looks like this:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-info href="data/x.xml"/>

</p:declare-step>
```

Result document:

```
<c:file xmlns:c="http://www.w3.org/ns/xproc-step"
        xml:base="file:/…/…/data/x.xml"
        name="x.xml"
        content-type="application/xml"
        readable="true"
        writable="true"
        hidden="false"
        last-modified="2025-02-05T12:05:59.36Z"
        size="88"/>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` option are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-info` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0134` (pg. 218)	It is a dynamic error if an implementation does not support `<p:file-info>` for a specified scheme.
`XC0135` (pg. 218)	It is a dynamic error if `<p:file-info>` is not available to the step due to access restrictions in the environment in which the pipeline is run.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.19 p:file-mkdir

Creates a directory.

Summary

```
<p:declare-step type="p:file-mkdir">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:file-mkdir` step creates the directory specified in the `href` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the created directory (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the directory to create.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port.

Description

The `p:file-mkdir` step creates the directory specified in the `href` option . The `result` port emits a small XML document with only a `<c:result>` element, containing the absolute URI of the created directory (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Any non-existent directories leading up to the final directory to create are created also.

Examples

Basic usage

The following example creates a `build` directory:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-mkdir href="build"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build</c:result>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` option are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-mkdir` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0114` (pg. 218)	It is a dynamic error if the directory referenced by the `href` option cannot be created.
`XC0140` (pg. 219)	It is a dynamic error if an implementation does not support `<p:file-mkdir>` for a specified scheme.
`XC0141` (pg. 219)	It is a dynamic error if `<p:file-mkdir>` not available to the step due to access restrictions in the environment in which the pipeline is run.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.20 p:file-move

Moves or renames a file or directory.

Summary

```
<p:declare-step type="p:file-move">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="target" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:file-move` step moves (or renames) a file or directory to a different location (or name).

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the target (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the source file or directory to move (or rename).
`target`	`xs:anyURI`	`true`		The URI of the target file or directory to move to.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port.

Description

The `p:file-move` step attempts to move (or rename) the file or directory specified in the `href` option to the location specified in the `target` option. The `result` port emits a small XML document with only a `<c:result>` element containing the absolute URI of the target (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

The inspiration for this step comes from the Unix `mv` command. If you're not used to it, it may come as a surprise that *moving* something can also mean *renaming* it:

- Moving `file:///a/b/c.xml` to `file:///a/b2/c.xml` *moves* the file to the `b2` directory. See also the Basic usage (pg. 61) example.
- But moving `file:///a/b/c.xml` to `file:///a/b/c2.xml` *renames* the file. See also the Renaming a file (pg. 61) example.

  An example of renaming a directory can be found in the example Copying a directory under a different name (pg. 51) in step `p:file-copy` (pg. 49).

If you're moving into another directory, this directory must exist. A target must not exist.

Examples

Basic usage

The following example moves a file `data/x1.xml` to `build/x1-copied.xml`:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-move href="data/x1.xml" target="build/x1-copied.xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/x1-copied.xml</c:result>
```

Please note that the target `build/` directory must exist, without an existing `x1-copied.xml` file.

Renaming a file

The following example *renames* an existing file `build/x1.xml` to `build/x2.xml`:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-move href="build/x1.xml" target="build/x2.xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/build/x2.xml</c:result>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` and `target` options are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-move` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0050` (pg. 216)	It is a dynamic error the file or directory cannot be copied to the specified location.
`XC0115` (pg. 218)	It is a dynamic error if the resource referenced by the `target` option is an existing file or other file system object.
`XC0148` (pg. 219)	It is a dynamic error if an implementation does not support `<p:file-move>` for a specified scheme.
`XC0149` (pg. 219)	It is a dynamic error if `<p:file-move>` is not available to the step due to access restrictions in the environment in which the pipeline is run.
`XC0158` (pg. 219)	It is a dynamic error if the `href` option names a directory, but the `target` option names a file.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.21 p:file-touch

Changes the modification timestamp of a file.

Summary

```
<p:declare-step type="p:file-touch">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="timestamp" as="xs:dateTime?" required="false" select="()"/>
</p:declare-step>
```

The `p:file-touch` step changes the modification timestamp of the file specified in the `href` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element containing the absolute URI of the modified file (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI of the file to change the modification timestamp of.
`fail-on-error`	`xs:boolean`	`false`	`true`	Determines what happens if an error occurs during the operation: • If this option is `true` (default), an appropriate XProc error is raised. • If this option is `false`, the step returns a `<c:error>` document (see here (https://spec.xproc.org/master/head/xproc/#err-vocab) for more information) on its `result` port.
`timestamp`	`xs:dateTime?`	`false`	`()`	If set, the file's modification timestamp is to this value. If absent or empty, the current system date/time is used.

Description

The `p:file-touch` step changes the modification timestamp of the file specified in the `href` option. The `result` port emits a small XML document with only a `<c:result>` element, containing the absolute URI of the changed file (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

If the file specified by the `href` option doesn't exist, an empty file will be created at the given location.

Examples

Basic usage

The following example changes the modification date of `data/x.xml` the current system date and time:
Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:file-touch href="data/x.xml"/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/data/x.xml</c:result>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- Relative values for the `href` option are resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:file-touch` is stored).
- Working on "normal" files and/or directories (on disk, URI scheme `file://`) is always supported. Whether any other types are supported is implementation-defined, and therefore depends on the XProc processor used. For this, also the interpretation/definition of what is a "directory" and "file" may vary.

Errors raised

Error code	Description
`XC0136` (pg. 218)	It is a dynamic error if an implementation does not support `<p:file-touch>` for a specified scheme.
`XC0137` (pg. 218)	It is a dynamic error if `<p:file-touch>` cannot be completed due to access restrictions in the environment in which the pipeline is run.
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.22 p:filter

Selects parts of a document.

Summary

```
<p:declare-step type="p:filter">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="true"/>
  <option name="select" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:filter` step selects parts of the source document based on a (possibly dynamically constructed) XPath select expression.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The source document to select the parts from.

`result`	`output`	`true`	`text`
			`xml`
			`html`
				`true`	The selected parts, as separate documents.

Options:

Name	Type	Req?	Description
`select`	`xs:string` (XPath expression)	`true`	The XPath expression that selects the parts.

Description

The `p:filter` step takes the XPath select expression in its `select` option, and with this select parts of the document appearing on the `source` port. The resulting document(s) (which might be zero, one or more) appear on the step's `result` port.

This step behaves just like adding a `select` attribute to an input port `<p:with-input>` element. What the `p:filter` step adds is the ability to construct the XPath expression dynamically. See Using a dynamic select expression (pg. 65) for an example of this.

Examples

Basic usage

The following example turns the single source document into a sequence of documents with information about bolts and pipes only.

```
<parts units="mm">
   <screw diameter="4"/>
   <bolt length="35"/>
   <pipe diameter="4"/>
</parts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" sequence="true"/>

  <p:filter select="/parts/(bolt | pipe)"/>

</p:declare-step>
```

Result documents:

```
<bolt length="35"/>
```

```
<pipe diameter="4"/>
```

Since the XPath expression for the `select` option is not dynamic, you don't actually need the `p:filter` step for this. The exact same result is achieved using a `select` attribute on a `<p:with-input>` element. For instance like below, using the `p:identity` (pg. 78) step:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" sequence="true"/>

  <p:identity>
    <p:with-input select="/parts/(bolt | pipe)"/>
  </p:identity>

</p:declare-step>
```

Using a dynamic select expression

Assume you are only interested in parts with a certain diameter. This diameter is passed to the pipeline using an option (in the example: `required-diameter`). The following pipeline dynamically constructs an XPath expression for this in the `p:filter` step `select` option. The input is the same as for the Basic usage (pg. 64) example.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" xmlns:xs="http://www.w3.org/2001/
XMLSchema" version="3.0" exclude-inline-prefixes="#all">

  <p:input port="source"/>
  <p:output port="result" sequence="true"/>

  <p:option name="required-diameter" as="xs:integer" select="4"/>

  <p:filter select="/parts/*[xs:integer(@diameter) eq {$required-diameter}]"/>

</p:declare-step>
```

Result documents:

```
<screw diameter="4"/>
```

```
<pipe diameter="4"/>
```

Remark: The `exclude-inline-prefixes="#all"` attribute on the pipeline root element is only there to prevent the `xs` namespace declaration showing up on the output documents. A superfluous namespace declaration doesn't matter but is unnecessary and looks sloppy.

Additional details

- No document-properties from the source document survive.
- The `base-uri` document-property of the documents appearing on the `result` port are the same as the base URI of their root element in the source document.

2.23 p:hash

Computes a hash code for a value.

Summary

```
<p:declare-step type="p:hash">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <option name="algorithm" as="xs:QName" required="true"/>
  <option name="value" as="xs:string" required="true"/>
  <option name="match" as="xs:string" required="false" select="'/*/node()'"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="version" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:hash` step generates a hash code for some value and injects this into the document appearing on the `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The source document to record the computed hash code *in*.

`result`	`output`	`true`	`text`
			`xml`
			`html`
				`false`	Result document, derived from the source document, intended to contain the hash code. See the `match` option and the description below.

Options:

Name	Type	Req?	Default	Description
`algorithm`	`xs:QName`	`true`		The hash computation algorithm to use. See the description below.
`value`	`xs:string`	`true`		The string value to calculate the hash code from.
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*/node()`	An XSLT selection pattern that tells `p:hash` where to insert the hash code in the source document. All node(s) matched are replaced with the computed hash code (the nodes themselves, not just their contents). If this option matches an attribute, the value of the attribute is changed. If this option matches the document-node `/`, the entire document is replaced with the computed hash code. The result will be a text document.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the hash code computation. Keys, values and their meaning are dependent on the XProc processor used.
`version`	`xs:string?`	`false`	`()`	Specifies the version of the hash computation algorithm used. If not specified, a default version is used, which depends on the value of the `algorithm` option. See the description below.

Description

A hash code in the digital world is a, relatively simple, value computed from some, possibly lengthy, input data. The same input data always results in the same hash code. Hash codes are also called hash values, (hash) digests, (digital) fingerprints, or simply hashes. Hash codes are used for various purposes, see for instance Wikipedia (https://en.wikipedia.org/wiki/Hash_function).

The `p:hash` step computes the hash code of the string value of the `value` option and inserts this into the document appearing on the `source` port. The result appears on the `result` port.

The algorithm used for computing the hash code *must* be specified using the `algorithm` option. There are 3 predefined values, that must be supported by all XProc processors:

`algorithm` option value	Default version	Algorithm used
`crc`	32	Cyclic Redundancy Check (https://en.wikipedia.org/wiki/Cyclic_redundancy_check)
`md`	5	Message-digest (https://en.wikipedia.org/wiki/MD5)
`sha`	1	Secure Hash Algorithm (https://en.wikipedia.org/wiki/SHA-1)

You can specify the algorithm version using the `version` option. If you don't specify this option and use one of the predefined `algorithm` option values, the default version from the table above is used.

Examples

Basic usage

The following example illustrates what happens when we compute a hash value and use the default value for the `match` option (`/*/node()`): all child nodes of the root element (in this example just the `<hash>` element) are replaced with the computed hash value.

Source document:

```
<hash-value>
   <hash>Will be replaced by the hash value!</hash>
</hash-value>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:hash algorithm="crc" value="Hi there!"/>

</p:declare-step>
```

Result document:

```
<hash-value>b5c57055b5c57055b5c57055</hash-value>
```

The following example shows the different values computed using the different standard algorithms. These are placed in attribute values.

Source document:

```
<hash-values crc="" md="" sha=""/>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:hash algorithm="crc" value="Hi there!" match="/*/@crc"/>
  <p:hash algorithm="md" value="Hi there!" match="/*/@md"/>
  <p:hash algorithm="sha" value="Hi there!" match="/*/@sha"/>

</p:declare-step>
```

Result document:

```
<hash-values crc="b5c57055"
             md="396199333edbf40ad43e62a1c1397793"
             sha="95e2b07e12754e52c37cfd485544d4f444597bff"/>
```

Hash code as text

If the `match` option matches the document-node `/`, the resulting document will be a text document containing the computed hash code only. Notice that in this case the input document doesn't matter.

Source document:

```
<anything/>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:hash algorithm="crc" value="Hi there!" match="/"/>

</p:declare-step>
```

Result text document:

```
b5c57055
```

Additional details

- `p:hash` preserves all document-properties of the document(s) appearing on its `source` port.
  The exception is when the `match` option matches the document-node `/`. In that case the resulting document will be of type text, the `content-type` document-property will become `text/plain` and a `serialization` document-property is removed. Any other document-property is preserved.
- If an XProc processor supports any other algorithm, its code (as supplied to the `algorithm` option) will be in a namespace.
- If the `match` option matches an attribute called `xml:base`, the base URI of this attribute's parent element is amended accordingly. This is a side-effect of changing an attribute with a pre-defined meaning (and in this case probably never useful).

Errors raised

Error code	Description
`XC0036` (pg. 216)	It is a dynamic error if the requested hash algorithm is not one that the processor understands or if the value or parameters are not appropriate for that algorithm.

2.24 p:http-request

Interact using HTTP (or related protocols).

Summary

```
<p:declare-step type="p:http-request">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <output port="report" primary="false" content-types="application/json" sequence="true"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="assert" as="xs:string" required="false" select="'.?status-code lt 400'"/>
  <option name="auth" as="map(xs:string, item()+)?" required="false" select="()"/>
  <option name="headers" as="map(xs:string, xs:string)?" required="false" select="()"/>
  <option name="method" as="xs:string?" required="false" select="'GET'"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:http-request` step allows pipelines to interact with resources (for instance websites) over HTTP or related protocols.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	Document(s) used in constructing the request body. By default, source documents are used for HTTP methods that require a body (for instance `POST`) only. If the HTTP method does not specify a body (for instance `GET`), any documents appearing on the `source` port are ignored. You can control this behaviour with the `send-body-anyway` parameter (see "Parameters" on page 72).
`result`	`output`	`true`	`any`	`true`	The request result document(s). See "The response result and report" on page 73.
`report`	`output`	`false`	`application/json`	`true`	A map containing information about the response. See "The response result and report" on page 73.

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI to use for the request.
`assert`	`xs:string`	`false`	`.?status-code lt 400`	Any request can fail, but what exactly failure is depends on the expectations of the receiver. This option takes an XPath expression that can inspect the request results. If the result of this expression (executed after a response is received) is `false`, dynamic error `XC0126` (pg. 78) is raised. See "Asserting the request status" on page 74
`auth`	`map(xs:string, item()+)?`	`false`	`()`	Information for the authentication of the request (in other words: about "logging in"). See "Request authentication" on page 71
`headers`	`map(xs:string, xs:string)?`	`false`	`()`	A map containing the request headers. Each map key is used as a header name and the value associated is used as the header value. There are some special rules regarding the request headers, see "Specifying request headers" on page 70. Request headers can influence the construction of the request. See "Usage of request headers" on page 71.
`method`	`xs:string?`	`false`	`GET`	The HTTP request method to use for the request. Its value is converted to upper-case. Any implementations must support the HTTP methods `GET` (default), `POST`, `PUT`, `DELETE`, and `HEAD`. Whether any other methods are supported is implementation-defined and therefore dependent on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	A map with parameters for fine-tuning the construction of the request and/or the handling of the server response. See "Parameters" on page 72.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	Before the document(s) on the `source` port are used, they are first serialized (as if written to disk). This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/), controlling this serialization. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence).

Description

The `p:http-request` allows you to send requests to some server and receive their response. You could use this, for instance, to access REST or other services on the web. Another use case is to have your pipeline play "web browser": get the contents of a web page and interpret this or fill in some page with a form in the background. Although the step is generic, it will probably be used most (exclusively?) using the HTTP(S) protocol, so we'll concentrate on this.

The HTTP(S) protocol is rather complex and, to support this, `p:http-request` is also complex. As a result of this, the description of `p:http-request` is long and may appear intimidating to users who are not familiar with the finer details of the HTTP(S) protocol. Luckily, simple interactions, like just requesting a single web page, are easy (see Basic usage (pg. 75)) Let's take it step by step (if there are parts you don't understand, chances are you don't need them). The HTTP(S) protocol itself is not explained, if you need more information about this, a good place to start would be on Wikipedia (https://en.wikipedia.org/wiki/HTTP).

1. The `p:http-request` step first constructs a request:
   - An HTTP(S) request is always to some URI (like `https://xprocref.org/`). You must specify this URI using the mandatory `href` option.
   - An HTTP(S) request has a method. Usual values are `GET`, `POST`, `PUT`, `DELETE`, and `HEAD`. You can specify this using the `method` option. Its default value is `GET`.
   - An HTTP(S) request has request headers: name/value pairs that contain additional information for the server. The main source for specifying request headers is the `headers` option. Some special handling applies, see "Specifying request headers" on page 70.
   - Once the request headers are known, some of this information is used by `p:http-request` for additional purposes, like determining the transfer encoding. See "Usage of request headers" on page 71 for more information.
   - Any documents that must accompany the request can be supplied on the `source` port.
   - Some interactions require authorization ("logging in"). This is usually specified with the `auth` option. See "Request authentication" on page 71 for more information.
   - It is possible to construct multipart request: requests where multiple documents are sent at once. See "Multipart requests" on page 74 for more information.
   - Further fine-tuning of the request is done using parameters specified in the `parameters` option. See "Parameters" on page 72 for more information.
2. The request is sent to the server and `p:http-request` waits for a response. How long the step will wait before giving up can be specified using parameters specified in the `parameters` option. See "Parameters" on page 72 for more information.
3. A response is received an interpreted:
   - Whether a response is considered successful can be specified using the `assert` option. If not, error `XC0126` (pg. 78) is raised. See "Asserting the request status" on page 74 for more information.
   - Further fine-tuning of the interpretation of the response is done using parameters specified in the `parameters` option. See "Parameters" on page 72 for more information.
   - Any documents contained in the response appear on the `result` port.
   - Additional information about the response (its response headers, status code, etc.) appears on the `report` port, as a map. See "The response result and report" on page 73 for more information.

Specifying request headers

An HTTP request has *request headers*: name/value pairs containing additional information for the server. See for instance Wikipedia (https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) for an overview.

The main source for `p:http-request` for constructing HTTP request headers is its `headers` option (see Viewing the request headers (pg. 75)).

If a *single* document appears on the `source` port and we're *not* constructing a multipart message, special rules apply:

- If the (single) document appearing on the `source` port:
  - is an XML, HTML or text document,
  - *and* has a `serialization` document-property,
  - *and* this `serialization` document-property has an entry called `encoding`,

  a `charset` is appended to the created `content-type` header of the HTTP request (for more information about this `charset` parameter, see for instance here (https://www.w3.org/International/articles/http-charset/index)).
- Any document-properties of the (single) document appearing on the `source` port that are in the `http://www.w3.org/ns/xproc-http` namespace will be added as request header, using their local name (their name without namespace) as the request header name.

  For a request parameter specified both in a document-property and in the map provided to the `headers` option, the one in the `headers` option takes precedence. This comparison is case-*in*sensitive.

When constructing a multipart message, not only the request itself but also its separate documents can have request headers. For more information on how this can be specified see "Multipart requests" on page 74.

Usage of request headers

Once the request headers are constructed (see "Specifying request headers" on page 70), some of these are used for additional purposes:

- If the value of the `content-type` request header starts with `multipart/`, a multipart request is constructed. (regardless of the number of documents appearing on the `source` port). See "Multipart requests" on page 74.
- For non multipart messages, it is possible to override the media type (`content-type` document-property) of the body document. If a single document appears on the `source` port, we're not constructing a multipart message and a `content-type` request header is specified, the value of the `content-type` request header overrides the value of the `content-type` document-property.
- If a `transfer-encoding` request header is present, the request is sent using that particular encoding (for more information about transfer encodings see here (https://www.rfc-editor.org/rfc/rfc9112#section-6.1)). Examples of values are `chunked`, `compress` or `gzip`.
- How authorization ("logging in") is done, is specified using the `authorization` request header. However, the `p:http-request` step also has an `auth` option for specifying this (see "Request authentication" on page 71). If this option is specified, the `authorization` request header, if present, is ignored. Instead, the value of the `authorization` request header is determined exclusively by the value of the `auth` option.

Request authentication

Information about the authorization of a request ("logging in") is sent to the server with the `authorization` request header. Experienced users could construct this request header themselves and add it to the step `headers` option (or use a document-property). However, in most cases, it is easier to pass the authorization information using the `auth` option and have `p:http-request` construct the `authorization` request header for you. If you use the `auth` option, any `authorization` request header passed in some other way is ignored.

The `auth` option contains the credentials (username, password, etc.) of the client and specifies what authentication method is used. It must be a map with string (`xs:string`) type keys. The following standard keys are defined:

Key	Value data type	Description
`username`	`xs:string`	The username for the request.
`password`	`xs:string`	The password associated with the username.
`auth-method`	`xs:string`	Specifies the authentication method to use. Standard values are `Basic` or `Digest` (see here (https://www.rfc-editor.org/info/rfc2617) for further information). Whether other authorization methods are supported and how to specify these is implementation-defined and therefore dependent on the XProc processor used.

Key	Value data type	Description
`send-authorization`	`xs:boolean`	This controls the "authorization challenge": • If this key is absent or its value is not `true`, a first request is sent *without* authorization information. If the server subsequently requests it, the request is resent *with* authorization information. • If this key's value is `true`, the first request immediately contains the authorization information.

If an authorization fails, the request is not retried.

Any other key/value pairs for the `auth` option map are implementation-defined and therefore dependent on the XProc processor used.

Parameters

The `parameters` option provides information for fine tuning the construction of the request and/or handling the response. It must be a map with string (`xs:string`) type keys. The following standard keys are defined:

Key	Value data type	Description
`override-content-type`	`xs:string`	The XProc processor must know how to interpret the body of a server response, its data type. Normally this is done by looking at the `content-type` response header. If this, for instance, is set to `application/xml`, the response body is interpreted as an XML document. Of course this must succeed, if not, error `XC0030` (pg. 78) is raised. If you specify an `override-content-type` parameter, its value is used instead of that in the `content-type` response header. The information about the `content-type` response header that appears on the `report` port (see "The response result and report" on page 73) is *not* changed and still reflects the actual value received from the server.
`http-version`	`xs:string`	Specifies the HTTP version to use for the request. Its default value is implementation-defined and therefore depends on the XProc processor used. Most probably it will be `1.1`.
`accept-multipart`	`xs:boolean`	If this parameter is present and has the value `false`, any multipart response will result in raising error `XC0125` (pg. 78). You can use this to prevent unexpected multipart responses wreak havoc in your pipeline.
`override-content-encoding`	`xs:string`	The XProc processor must know how the encoding of a server response (for instance: `utf-8`). Normally this is done by looking at the `content-encoding` response header. If you specify an `override-content-encoding` parameter, its value is used instead of that in the `content-encoding` response header. The information about the `content-encoding` response header that appears on the `report` port (see "The response result and report" on page 73) is *not* changed and still reflects the actual value received from the server.
`permit-expired-ssl-certificate`	`xs:boolean`	If this parameter is present and has the value `true`, `p:http-request` does *not* reject a response where the server provides an expired SSL certificate.
`permit-untrusted-ssl-certificate`	`xs:boolean`	If this parameter is present and has the value `true`, `p:http-request` does *not* reject a response where the server provides an SSL certificate which is not trusted, for example, because the certificate authority (CA) is unknown.

Key	Value data type	Description
`follow-redirect`	`xs:integer`	Sometimes a server responds with a *redirect*, meaning something like "please repeat the request to this different URI". The `follow-redirect` parameter tells the XProc processor what to do when a redirect is received: • If its value is `0`, redirects are not followed. • If its value is `-1`, redirects are followed indefinitely. • If its value is *positive*, at most this number of subsequent redirects are followed. The default behaviour, when the `follow-redirect` parameter is not present, is implementation-defined and therefore dependent on the XProc processor used.
`timeout`	`xs:integer`	Specifies the number of seconds to wait for a response. If no response is received after approximately this number of seconds, the request is terminated and HTTP status `408` is assumed.
`fail-on-timeout`	`xs:boolean`	Sometimes a request results in a timeout. This can either happen by receiving a response with HTTP status `408` or because the number of seconds specified in the `timeout` parameter is exceeded. If a `fail-on-timeout` parameter is present and has the value `true`, this will result in raising error `XC0078` (pg. 78). This might be confusing, because XProc also has a `[p:]timeout` attribute, useable on all steps, that tells the XProc processor how long a step invocation is allowed to take (also specified in seconds). What happens depends on whatever comes first: • If the number of seconds specified in the `[p:]timeout` attribute is exceeded, error `XD0053` is raised (a generic timeout error). Be careful when you want to use this: whether a processor supports timeouts using the `[p:]timeout` attribute, and if it does, how precisely and precisely how the execution time of a step is measured, is implementation-defined and therefore dependent on the XProc processor used. • If `fail-on-timeout` is `true` and a timeout happens, error `XC0078` (pg. 78) is raised.
`status-only`	`xs:boolean`	If this parameter is present and its value is `true`, it indicates that the pipeline author is interested in the response code only. The `result` port will not emit anything. The map on the `report` port will return an empty map as value of its `headers` entry.
`suppress-cookies`	`xs:boolean`	If this parameter is present and its value is `true`, no cookies are sent with the request.
`send-body-anyway`	`xs:boolean`	By default, whether a body is sent with the request depends on the HTTP method used (the value of the `method` option). For instance, the `GET` method does not specify a body. When the `GET` method is used, by default any document(s) on the `source` port are ignored. When the `send-body-anyway` parameter is present and its value is `true`, a request body will always be constructed, even if the HTTP method used does not specify this.

Any other key/value pairs for the `parameters` option map are implementation-defined and therefore dependent on the XProc processor used.

The response result and report

When an answer is received from the server, document(s) in the response body will appear as document(s) on the `result` port. Each document will be parsed according to its content-type. You can override this behaviour using the `override-content-type` parameter (see "Parameters" on page 72).

In case of a multipart response, each part will become a separate document appearing on the `result` port. Any response headers associated with a specific part are added to the document-properties of the resulting document.

The `report` port always returns a map with the following keys/entries:

Key	Value data type	Description
`status-code`	`xs:integer`	The HTTP status code for the request, for instance `200` (success) or `404` (failure).
`base-uri`	`xs:anyURI`	The URI of the request. In case of HTTP redirection, this value may be different from the original request URI.
`headers`	`map(xs:string, xs:string)`	The HTTP headers returned for the request. Header names are in lower-case. The map may be empty.

Asserting the request status

Any request can fail, but what exactly failure is depends on the expectations of the receiver. The `assert` option of `p:http-request` takes an XPath expression that inspects the request results:

- It must contain a valid (boolean) XPath expression.
- This expression will be executed when a response is received.
- The context item when executing the expression is the map that also appears on the `report` port (see "The response result and report" on page 73).
- If the expression evaluates to `false`, the request is considered failed and error `XC0126` (pg. 78) is raised.
- If the expression evaluates to `true`, the request is considered successful. No error is raised.

The default value for the `assert` option is `.?status-code lt 400`. Since the context item is the map on the `report` port, the dot operator `.` here refers to this map. The `.?status-code` part is one of the ways to access a map entry (another way to write this is `.('status-code')`). The referred map entry contains the received (integer) HTTP status code. According to its default interpretation, when less than `400`, the response is considered a success. If it's greater than or equal to `400`, it is considered a failure and error `XC0126` (pg. 78) is raised.

Multipart requests

Multipart requests combine one or more sets of data into a single HTTP request. You use this for file uploads and/or transferring data of several types in one go. For instance, a web page that allows you to upload several images could use a single multipart request to sent all these images to the server. For more information, see for instance Wikipedia (https://en.wikipedia.org/wiki/MIME#Multipart_messages) or the W3C multipart protocol description (https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html).

The `p:http-request` step constructs a multipart request if one or both of the following conditions is met:

- Multiple documents appear on the `source` port.
- The `content-type` request header (see "Specifying request headers" on page 70) starts with `multipart/`.

If no specific `content-type` request header is specified and the `source` receives multiple documents, the content type is set to `multipart/mixed`.

Multipart request must have a *boundary marker*: a string of characters that is inserted in between the message parts. This is critical, because this marker must *not* appear anywhere in the data itself. If it does, the request is considered malformed. The boundary marker as used by `p:http-request` is constructed as follows:

- If the `content-type` request header contains a boundary parameter, this is used.

  For instance, by setting the `content-type` to `multipart/mixed; boundary=gc0p4Jq0M2Yt08jU534c0p`, the boundary marker becomes `--gc0p4Jq0M2Yt08jU534c0p` (the two hyphens in front are prescribed by the protocol).
- If this is not the case, the boundary marker is implementation-defined and therefore dependent on the XProc processor used. Unfortunately, there is no guarantee this boundary marker does not appear in the data itself (which would make the request malformed).

When constructing the multipart message, each document on the `source` port is serialized (as if written to disk). If a document has a `serialization` document-property, this is used to determine the serialization format.

The separate documents in a multipart message can have request headers on their own. Examples of often used headers are `id`, `description` and `disposition`. Document-properties of documents on the `source` port that are in the `http://www.w3.org/ns/xproc-http` namespace will be used to construct request headers for that particular document, using their local name (their name without namespace) as the request header name.

Examples

Basic usage

The following example:

- Uses `p:http-request` to ask for the home page of the https://xprocref.org website, just like a web browser. This fires an HTTPS `GET` request and waits for the answer.

  Notice that we have to supply a value for the `source` port, even if we don't need it. In this case we simply set it to `<p:empty>`.
- We strip the resulting HTML page to just its `<head>` (otherwise the result would be too big to display).

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result" sequence="true"/>

  <p:http-request href="https://xprocref.org">
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:http-request>

  <p:delete match="/h:html/h:body"/>

</p:declare-step>
```

Resulting HTML fragment:

```
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
      <meta name="viewport" content="width=device-width, initial-scale=1"/>
      <meta name="keywords" content="xproc xprocref xml"/>
      <link href="css/bootstrap.min.css" rel="stylesheet"/>
      <link href="css/xprocref.css" rel="stylesheet"/>
      <script defer="" src="js/bootstrap.bundle.min.js"/>
      <title>XProc steps (3.1)</title>
      <link rel="shortcut icon" href="images/favicon.ico"/>
   </head>
</html>
```

Viewing the request headers

When we want to see what `p:http-request` sends as request headers, we need some developer website that tells us what the request headers are. There is such a server by Beeceptor (https://beeceptor.com/): send some HTTP request to `https://echo.free.beeceptor.com` and what you receive is a JSON message containing information about your request.

Sending a simple `GET` request to this URI returns (results vary depending on your IP, operating system, browser, etc.):

```
{
  "method": "GET",
  "protocol": "https",
  "host": "echo.free.beeceptor.com",
  "path": "/",
  "ip": "84.29.5.211:52418",
  "headers": {
    "Host": "echo.free.beeceptor.com",
    "User-Agent": "Apache-HttpClient/4.5.10 (Java/17.0.12)",
    "Accept": "*/*",
    "Accept-Encoding": "gzip,deflate"
  },
  "parsedQueryParams": {}
}
```

The following example sends a simple HTTPS request to this server and uses the resulting JSON to construct an XML document showing the HTTP request headers sent:

- The invocation of `p:http-request` just sends a `GET` request to `https://echo.free.beeceptor.com`.
- The result is a JSON message that the XProc processor turns into a map. This is now our context item, accessible with the dot operator `.`
- A sub-map in this map called `headers` contains the header information we're interested in. We extract this part into a variable `$headers`.
- The `<p:for-each>` loops over all keys in the `$headers` sub-map.
- A `p:identity` (pg. 78) step is used to construct a (single element) XML document, containing the request header name and value: `<request-header name="…" value="…"/>`
- The `<p:for-each>` loop now emits a sequence of documents, one for each request header. A `p:wrap-sequence` (pg. 187) step wraps this into an `<http-request-headers>` root element to produce a well-formed XML document.

```
<p:declare-step xmlns:map="http://www.w3.org/2005/xpath-functions/map" xmlns:p="http://www.w3.org/ns/
xproc" version="3.0">

  <p:output port="result" sequence="true"/>

  <p:http-request href="https://echo.free.beeceptor.com">
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:http-request>

  <p:variable name="headers" as="map(*)" select=".?headers"/>
  <p:for-each>
    <p:with-input select="map:keys($headers)"/>
    <p:identity>
      <p:with-input>
        <request-header name="{.}" value="{$headers(.)}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="http-request-headers"/>

</p:declare-step>
```

Result document:

```
<http-request-headers>
   <request-header name="Host" value="echo.free.beeceptor.com"/>
   <request-header name="User-Agent" value="Apache-HttpClient/4.5.10 (Java/17.0.11)"/>
   <request-header name="Accept" value="*/*"/>
   <request-header name="Accept-Encoding" value="gzip,deflate"/>
</http-request-headers>
```

Adding a request header

The `headers` option can be used for additional request headers. In this example we add the bogus request header called `xyz` and set it to the value `123`. The code to view the request headers is identical to that of Viewing the request headers (pg. 75):

```
<p:declare-step xmlns:map="http://www.w3.org/2005/xpath-functions/map" xmlns:p="http://www.w3.org/ns/
xproc" version="3.0">

  <p:output port="result" sequence="true"/>

  <p:http-request href="https://echo.free.beeceptor.com" headers="map{'xyz': '123' }">
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:http-request>

  <p:variable name="headers" as="map(*)" select=".?headers"/>
  <p:for-each>
    <p:with-input select="map:keys($headers)"/>
    <p:identity>
      <p:with-input>
        <request-header name="{.}" value="{$headers(.)}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="http-request-headers"/>

</p:declare-step>
```

Result document:

```
<http-request-headers>
   <request-header name="Host" value="echo.free.beeceptor.com"/>
   <request-header name="User-Agent" value="Apache-HttpClient/4.5.10 (Java/17.0.11)"/>
   <request-header name="Accept" value="*/*"/>
   <request-header name="Accept-Encoding" value="gzip,deflate"/>
   <request-header name="Xyz" value="123"/>
</http-request-headers>
```

Notice that the name of the request header is capitalized into `Xyz`. Request header names are case-insensitive, but it is custom to capitalize them (start with an upper-case character).

Viewing the response headers

Inspecting the response headers can be done by using the map returned on the `report` port (see "The response result and report" on page 73). The code to view the response headers is almost identical to that of Viewing the request headers (pg. 75):

```
<p:declare-step xmlns:map="http://www.w3.org/2005/xpath-functions/map" xmlns:p="http://www.w3.org/ns/
xproc" version="3.0">

  <p:output port="result" sequence="true"/>

  <p:http-request href="https://echo.free.beeceptor.com" name="request">
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:http-request>

  <p:variable name="response-headers" as="map(*)" select=".?headers" pipe="report@request"/>
  <p:for-each>
    <p:with-input select="map:keys($response-headers)"/>
    <p:identity>
      <p:with-input>
        <response-header name="{.}" value="{$response-headers(.)}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="http-response-headers"/>

</p:declare-step>
```

Result document:

```
<http-response-headers>
   <response-header name="access-control-allow-origin" value="*"/>
   <response-header name="alt-svc" value="h3=&#34;:443&#34;; ma=2592000"/>
   <response-header name="content-type" value="application/json"/>
   <response-header name="date" value="Wed, 25 Jun 2025 11:24:53 GMT"/>
   <response-header name="vary" value="Accept-Encoding"/>
   <response-header name="transfer-encoding" value="chunked"/>
</http-response-headers>
```

Additional details

- A relative value for the `href` option is resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:http-request` is stored). This is very probably not what you want.
- HTTP request header names are case-insensitive, but keys in maps are not. This means that you could specify the same request header multiple times in the `headers` option. For instance as `Content-Type` and `content-type`. If that happens, error `XC0127` (pg. 78) is raised.
- When constructing multipart requests (see "Multipart requests" on page 74): multiple documents on the `source` port combined with a `content-type` header that does *not* start with `multipart/` raises error `XC0133` (pg. 78).

Errors raised

Error code	Description
`XC0003` (pg. 216)	It is a dynamic error if a "`username`" or a "`password`" key is present without specifying a value for the "`auth-method`" key, if the requested `auth-method` isn't supported, or the authentication challenge contains an authentication method that isn't supported.
`XC0030` (pg. 216)	It is a dynamic error if the response body cannot be interpreted as requested (e.g. `application/json` to override `application/xml` content).
`XC0078` (pg. 217)	It is a dynamic error if the value associated with the "`fail-on-timeout`" is associated with `true()` and a HTTP status code `408` is encountered.
`XC0122` (pg. 218)	It is a dynamic error if the given method is not supported.
`XC0123` (pg. 218)	It is a dynamic error if any key in the "`auth`" map is associated with a value that is not an instance of the required type.
`XC0124` (pg. 218)	It is a dynamic error if any key in the "parameters" map is associated with a value that is not an instance of the required type.
`XC0125` (pg. 218)	It is a dynamic error if the key "`accept-multipart`" as the value `false()` and a multipart response is detected.
`XC0126` (pg. 218)	It is a dynamic error if the XPath expression in `assert` evaluates to `false`.
`XC0127` (pg. 218)	It is a dynamic error if the `headers` map contains two keys that are the same when compared in a case-insensitive manner.
`XC0128` (pg. 218)	It is a dynamic error if the URI's scheme is unknown or not supported.
`XC0129` (pg. 218)	It is a dynamic error if the requested HTTP version is not supported.
`XC0131` (pg. 218)	It is a dynamic error if the processor cannot support the requested encoding.
`XC0132` (pg. 218)	It is a dynamic error if the override content encoding cannot be supported.
`XC0133` (pg. 218)	It is a dynamic error if more than one document appears on the `source` port and a `content-type` header is present and the content type specified is not a multipart content type.
`XC0203` (pg. 220)	It is a dynamic error if the specified boundary is not valid (for example, if it begins with two hyphens "--").
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.25 p:identity

Copies the source to the result without modifications.

Summary

```
<p:declare-step type="p:identity">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
</p:declare-step>
```

The `p:identity` step makes a verbatim copy of what appears on its `source` port to its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The source document(s)
`result`	`output`	`true`	`any`	`true`	The resulting document(s). These will be exactly the same as what appeared on the `source` port.

Description

The `p:identity` step does… nothing. It makes a verbatim copy of all documents appearing on its `source` port to its `result` port. Although it doesn't do anything, it is actually extremely useful and virtually indispensable. The examples below show some use cases.

Examples

Create a fixed document

There are many situations where you need to create a fixed document in your pipeline. For instance:

- On an error catch you want some `<error …>` element as result:

```
<p:catch>
  <p:identity>
    <p:with-input>
      <error … />
    </p:with-input>
  </p:identity>
</p:catch>
```

- Some pipelines write their main results to disk and the actual output of the pipeline doesn't matter. In these cases it is often useful to produce some kind of report document with relevant information (for instance, when it happened, where the results are, etc.):

```
<p:identity>
  <p:with-input>
    <report timestamp="{current-dateTime()}" href-result="{$href-result-location}" … />
  </p:with-input>
</p:identity>
```

Create an explicit anchor points in your pipeline

Because the `p:identity` step does't do anything, it can be used to create "anchor points" in your pipeline. Assume you have a complicated pipeline where some version of the document flowing through must be used somewhere else. The `p:identity` step can be used to mark such a location explicitly. In the following example an anchor point called `raw-version` is created and, later on, referred to:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <!-- Some complicated computations… -->

  <p:identity name="raw-version"/>

  <!-- Some more complicated computations… -->

  <!-- And then a step refers back to the raw version: -->
  <p:insert match="/*" position="first-child">
    <p:port port="insertion" pipe="@raw-version"/>
  </p:insert>

  <!-- And some more computations… -->

</p:declare-step>
```

You could also achieve this by using the `name="raw-version"` attribute on the last step of the first batch of computations. However, by using an explicit `p:identity` step it stands out in the code. Also, when the computations change (and they will), you don't have to remember to keep the `name="raw-version"` attribute on the *last* one always.

Produce a processing message

XProc has a `message` attribute (or `p:message` on steps not in the XProc namespace) that results in a message when the pipeline runs. Where this message appears depends on how the pipeline is run. Sometimes you want to explicitly produce messages when some point in your pipeline is reached. Since `p:identity` does't do anything, it is ideal for this:

```
<p:identity message="We started processing!"/>
<p:identity message="- Input document {$href-input}"/>
<p:identity message="- Processing type {$processing-type}"/>
```

Additional details

- `p:identity` preserves all document-properties of the document(s) appearing on its `source` port.

2.26 p:insert

Inserts one document into another.

Summary

```
<p:declare-step type="p:insert">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html text" sequence="false"/>
  <input port="insertion" primary="false" content-types="xml html text" sequence="true"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
  <option name="position" as="xs:string" required="false" select="'after'" values="('first-child','last-
child','before','after')"/>
</p:declare-step>
```

The `p:insert` step inserts the document(s) appearing on the `insertion` port into the document appearing on the `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The base document to insert *in*.
`result`	`output`	`true`	`xml` `html` `text`	`false`	The resulting document
`insertion`	`input`	`false`	`xml` `html` `text`	`true`	The document(s) to insert.

Options:

Name	Type	Req?	Default	Description
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern that indicates where to insert.
`position`	`xs:string`	`false`	`after`	Where to insert, relative to what was matched by the `match` option. See "The position option" on page 80.

Description

The `p:insert` step can be used to insert one or more documents into another. It searches the document appearing on its `source` port for matches as indicated by the `match` option. It then inserts the document(s) appearing on its `insertion` port, as indicated by the `position` option. The result of the merge is emitted on the `result` port.

The position option

The `position` option tells `p:insert` where to insert the document(s) in the `insertion` port, relative to what was matched by the `match` option. There are 4 possible values:

Value	Description
`first-child`	The insertion is made as the *first child* of what was matched.
`last-child`	The insertion is made as the *last child* of what was matched.
`before`	The insertion is made directly *before* what was matched. In other words: the insertion becomes the immediate preceding sibling of the match.
`after`	The insertion is made directly *after* what was matched. In other words: the insertion becomes the immediate following sibling of the match.

Examples

Basic usage

This simple example shows how to insert a (for the example, fixed) document into another, as a first child of a match:

Source document:

```
<things>
   <thing id="123">USB adapter</thing>
   <thing id="456">Joystick</thing>
   <thing id="789">Mouse</thing>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert position="first-child">
    <p:with-input port="insertion">
      <thing id="999">Keyboard</thing>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Result document:

```
<things>
   <thing id="999">Keyboard</thing>
   <thing id="123">USB adapter</thing>
   <thing id="456">Joystick</thing>
   <thing id="789">Mouse</thing>
</things>
```

Or, using the same source document, as the its last child:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert position="last-child">
    <p:with-input port="insertion">
      <thing id="999">Keyboard</thing>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Result document:

```
<things>
   <thing id="123">USB adapter</thing>
   <thing id="456">Joystick</thing>
   <thing id="789">Mouse</thing>
   <thing id="999">Keyboard</thing>
</things>
```

The following example, again using the same source document, inserts a new thing using the `before` value of the `position` option:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert match="/things/thing[@id eq '456']" position="before">
    <p:with-input port="insertion">
      <thing id="999">Keyboard</thing>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Result document:

```
<things>
   <thing id="123">USB adapter</thing>
   <thing id="999">Keyboard</thing>
   <thing id="456">Joystick</thing>
   <thing id="789">Mouse</thing>
</things>
```

Inserting multiple times

The following example shows that when the `match` option matches multiple times, the insertion occurs multiple times:

Source document:

```
<things>
   <thing id="123">
      <name>USB adapter</name>
   </thing>
   <thing id="456">
      <name>Joystick</name>
   </thing>
   <thing id="789">
      <name>Mouse</name>
   </thing>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert match="thing" position="last-child">
    <p:with-input port="insertion">
      <description>TBD</description>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Result document:

```
<things>
   <thing id="123">
      <name>USB adapter</name>
      <description>TBD</description>
   </thing>
   <thing id="456">
      <name>Joystick</name>
      <description>TBD</description>
   </thing>
   <thing id="789">
      <name>Mouse</name>
      <description>TBD</description>
   </thing>
</things>
```

Inserting text

Starting XProc version 3.1, it is also possible to insert text into a document using the `p:insert` step:

Source document:

```
<things>
   <thing id="123">USB adapter</thing>
   <thing id="456">Joystick</thing>
   <thing id="789">Mouse</thing>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:insert match="/things/thing[@id eq '456']" position="last-child">
    <p:with-input port="insertion">
      <p:inline content-type="text/plain"> (special!)</p:inline>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Result document:

```
<things>
   <thing id="123">USB adapter</thing>
   <thing id="456">Joystick (special!)</thing>
   <thing id="789">Mouse</thing>
</things>
```

Additional details

- It must be possible to insert the document(s) at the indicated location(s). For instance, the `match` option cannot match an attribute and you cannot insert something before or after the document node. If this happens, an appropriate error is raised.
- `p:insert` preserves all document-properties of the document(s) appearing on its `source` port.

  The document-properties of the document(s) appearing on its `insertion` port are not used/preserved.
- If the `match` option matches multiple times, multiple instances of the document(s) on the `insertion` port are inserted. See Inserting multiple times (pg. 82).
- If the `insertion` port receives no document(s), nothing happens. The step will act as a `p:identity` (pg. 78) step.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.
`XC0024` (pg. 216)	It is a dynamic error if the selection pattern matches a document node and the value of the `position` is "`before`" or "`after`".
`XC0025` (pg. 216)	It is a dynamic error if the selection pattern matches anything other than an element or a document node and the value of the `position` option is "`first-child`" or "`last-child`".

2.27 p:invisible-xml

Performs invisible XML processing.

Summary

```
<p:declare-step type="p:invisible-xml">
  <input port="source" primary="true" content-types="any -xml -html" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <input port="grammar" primary="false" content-types="text xml" sequence="true"/>
  <option name="fail-on-error" as="xs:boolean" required="false" select="true()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:invisible-xml` step parses the document on the `source` port using invisible XML (https://invisiblexml.org/1.0/). The grammar for this must be provided on the `grammar` port. The result will be emitted on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
source	input	true	any - xml - html	false	The source document to parse using the invisible XML grammar (https://invisiblexml.org/1.0/) provided on the grammar port. If the grammar port is empty, this must contain a valid invisible XML grammar (https://invisiblexml.org/1.0/). See the description of the grammar port.
result	output	true	any	true	The result of parsing the document on the source port.
grammar	input	false	text xml	true	One of the following: • A single document containing the invisible XML grammar (https://invisiblexml.org/1.0/) to use for parsing the document on the source port. This grammar can either be in text or XML format. • If empty, the document on the source must contain a valid invisible XML grammar (https://invisiblexml.org/1.0/). This is converted to its XML representation and returned on the result port. See Parsing the invisible XML grammar (pg. 85) for an example.

Options:

Name	Type	Req?	Default	Description
fail-on-error	xs:boolean	false	true	Determines what happens if the document cannot be parsed: • If true, error XC0205 (pg. 86) is raised. • If false, the step always succeeds. The invisible XML specification (https://invisiblexml.org/1.0/) provides a mechanism to identify failed parses in the output.
parameters	map(xs:QName, item()*)?	false	()	Parameters used to control the parsing. The XProc specification does not define any parameters for this option. A specific XProc processor (or parser used) might define its own.

Description

Invisible XML (or ixml) is a method for treating non-XML documents as if they were XML, enabling authors to write documents and data in a format they prefer while providing XML for processes that are more effective with XML content.

The p:invisible-xml takes a document, usually text, and parses this using an invisible XML grammar (https://invisiblexml.org/1.0/) into an XML document. The grammar must be provided on the grammar port. The result will appear on the result port.

Invisible XML has both a text and an XML representation and you can use both representations on the grammar port. Converting the text to the XML grammar can be done by leaving the grammar port empty and providing the text based grammar on the source port. See Parsing the invisible XML grammar (pg. 85) for an example.

In most cases, p:invisible-xml relies on an external parser. You'll probably have to do some XProc processor dependent configuration before this step will work. Please consult the XProc processor documentation about this.

Examples

Basic usage

We're going to use a very basic invisible XML grammar (https://invisiblexml.org/1.0/) that parses a written date into XML. The grammar looks like this:

```
date: s?, day, s, month, (s, year)? .
-s: -" "+ .
day: digit, digit? .
-digit: "0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9".
month: "January"; "February"; "March"; "April";
       "May"; "June"; "July"; "August";
       "September"; "October"; "November"; "December".
year: (digit, digit)?, digit, digit .
```

The input document is:

```
31 December 2021
```

Using the `p:invisible-xml` step to parse this, the result is as follows:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:invisible-xml>
    <p:with-input port="grammar" href="grammar.txt"/>
  </p:invisible-xml>

</p:declare-step>
```

Result document:

```
<date>
   <day>31</day>
   <month>December</month>
   <year>2021</year>
</date>
```

Parsing the invisible XML grammar

We can parse the text representation of an invisible XML grammar into its XML representation by leaving the `grammar` port empty and provide the text grammar on the `source` port. Using the same grammar as in Basic usage (pg. 85), the result is:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:invisible-xml>
    <p:with-input port="grammar">
      <p:empty/>
    </p:with-input>
  </p:invisible-xml>

</p:declare-step>
```

Result document:

```
<ixml>
   <rule name="date">
      <alt>
         <option>
            <nonterminal name="s"/>
         </option>
         <nonterminal name="day"/>
         <nonterminal name="s"/>
         <nonterminal name="month"/>
         <option>
            <alts>
               <alt>
                  <nonterminal name="s"/>
                  <nonterminal name="year"/>
               </alt>
            </alts>
         </option>
      </alt>
   </rule>
   <rule mark="-" name="s">
      <alt>
         <repeat1>
            <literal tmark="-" string=" "/>
         </repeat1>
      </alt>
   </rule>
   <rule name="day">
      <alt>
         <nonterminal name="digit"/>
         <option>
            <nonterminal name="digit"/>
         </option>
      </alt>
   </rule>
   <!-- … (shortened) -->
</ixml>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- The resulting document will in the vast majority of cases be XML. However, the implementation allows for returning other document types. If, how and when this happens is implementation defined and therefore dependent on the XProc processor used.

Errors raised

Error code	Description
`XC0205` (pg. 220)	It is a dynamic error if the source document cannot be parsed by the provided grammar.
`XC0211` (pg. 220)	It is a dynamic error if more than one document appears on the `grammar` port.
`XC0212` (pg. 220)	It is a dynamic error if the grammar provided is not a valid Invisible XML grammar.

2.28 p:json-join

Joins documents into a JSON array document.

Summary

```
<p:declare-step type="p:json-join">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="application/json" sequence="false"/>
  <option name="flatten-to-depth" as="xs:string?" required="false" select="'0'"/>
</p:declare-step>
```

The `p:json-join` step joins the document(s) appearing on the `source` port into a JSON array. This array is returned as a single JSON document on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The documents to join.
`result`	`output`	`true`	`application/json`	`false`	The resulting JSON array document containing the source documents.

Options:

Name	Type	Req?	Default	Description
`flatten-to-depth`	`xs:string?`	`false`	`0`	Specifies whether and to which depth JSON source documents that are arrays must be "flattened". See "Flattening arrays" on page 87. Please notice that the data type of this option is `xs:string` (and not `xs:integer` as you might expect). This is because it can also take the value `unbounded`.

Description

The `p:json-join` step takes the document(s) appearing on its `source` port and joins them into a JSON array. How the source documents end up in the resulting array depends on their type:

- A JSON source document that is *not* an array is added to the result as a member of the resulting array.
- A JSON source document that is itself an array is also added to the resulting array. However, depending on the value of the `flatten-to-depth` option, additional "flattening" can happen. See "Flattening arrays" on page 87.
- An XML, HTML or text document is first serialized (as if written to disk) and then added as a string to the resulting array.
- Whether `p:json-join` can handle any other media types is implementation-defined and therefore dependent on the XProc processor used.

The resulting array is emitted as a single JSON document on the `result` port.

Flattening arrays

Depending on the value of the `flatten-to-depth` option, "flattening" of the resulting array takes place. Flattening here means that if there are members of the resulting array that are itself arrays, their members will appear as individual members in the final result. For instance, flattening `["a", "b", ["c", "d"] ]` results in `["a", "b", "c", "d"]`.

What happens exactly depends on the value of the `flatten-to-depth` option:

- If `0` (default), no flattening takes place.
- If `1`, any array on the `source` port is flattened into the final result.
- If greater than `1`, flattening is applied recursively to arrays in arrays, up to the given depth.
- If `unbounded`, all arrays are flattened.

See Flattening arrays (pg. 88) for an example.

Examples

Basic usage

The following pipeline produces a sequence of 3 documents on the `source` port of `p:json-join`: An XML, a text and a JSON document. This is merrily joined into a single array:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline>
      <some-xml a="b"/>
    </p:inline>
    <p:inline content-type="text/plain">Hello there!</p:inline>
    <p:inline content-type="application/json" expand-text="false">{"key": 12345}</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-join/>

</p:declare-step>
```

Result document:

```
["\n      <some-xml a=\"b\"\/>\n    ","Hello there!",{"key":12345}]
```

Flattening arrays

The following pipeline produces a sequence of 2 documents on the `source` port of `p:json-join`: A JSON text and a JSON array. If we do *not* flatten it, the source array will simply appear as a member of the resulting array:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline content-type="application/json">"Hello!"</p:inline>
    <p:inline content-type="application/json">["a", "b", ["c", "d"] ]</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-join/>

</p:declare-step>
```

Result document:

```
["Hello!",["a","b",["c","d"]]]
```

By setting the `flatten-to-depth` option to `1`, the source array is flattened:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline content-type="application/json">"Hello!"</p:inline>
    <p:inline content-type="application/json">["a", "b", ["c", "d"] ]</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-join flatten-to-depth="1"/>

</p:declare-step>
```

Result document:

```
["Hello!","a","b",["c","d"]]
```

If we increase the `flatten-to-depth` option to `2`, the inner array is also flattened. A value `unbounded` would have had the same effect here.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline content-type="application/json">"Hello!"</p:inline>
    <p:inline content-type="application/json">["a", "b", ["c", "d"] ]</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-join flatten-to-depth="2"/>

</p:declare-step>
```

Result document:

```
["Hello!","a","b","c","d"]
```

Additional details

- No document-properties of the source document(s) survive.
- The resulting document has no `base-uri` property.
- If there are no documents appearing on the `source` port, the `result` port returns the empty sequence.

Errors raised

Error code	Description
`XC0111` (pg. 218)	It is a dynamic error if a document of an unsupported document type appears on port `source` of `p:json-join`.
`XC0119` (pg. 218)	It is a dynamic error if `flatten` is neither "`unbounded`", nor a string that may be cast to a non-negative integer.

2.29 p:json-merge

Joins documents into a JSON map document.

Summary

```
<p:declare-step type="p:json-merge">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="application/json" sequence="false"/>
  <option name="duplicates" as="item()*" required="false" select="'use-first'" values="('reject', 'use-
first', 'use-last', 'use-any', 'combine')"/>
  <option name="key" as="xs:string" required="false" select="'concat("_",$p:index)'"/>
</p:declare-step>
```

The `p:json-merge` step merges the document(s) appearing on the `source` port into a JSON map. This map is returned as a single JSON document on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The documents to join.
`result`	`output`	`true`	`application/json`	`false`	The resulting JSON map document containing the source documents.

Options:

Name	Type	Req?	Default	Description
`duplicates`	`item()*`	`false`	`use-first`	Specifies what to do with duplicate keys in the resulting map. See "Handling duplicates" on page 90.
`key`	`xs:string` (XPath expression)	`false`	`concat("_", $p:index)`	An XPath expression that computes the value for the key of an entry in the resulting map. This expression is evaluated with the document it is applied to as context item. A variable `$p:index` is available that holds the index (sequence number) of the document on the `source` port. See Computing a different key (pg. 91) for an example of how to use this option.

Description

The `p:json-merge` step takes the document(s) appearing on its `source` port and joins them into a JSON map (also know as JSON object). How the source documents end up in the resulting map depends on their type:

- For a JSON source document that is a map, all key/value pairs are copied into the result map.
- For JSON documents that are not a map, XML, HTML and text documents, a new key/value pair is created. The key is computed using the XPath expression in the `key` option. Regarding their value:
  - A JSON document is used as is.
  - An XML, HTML or text document is first serialized (as if written to disk) and then added as a string to the resulting array.
- Whether `p:json-merge` can handle any other media types is implementation-defined and therefore dependent on the XProc processor used.

Handling duplicates

While filling up the result map, it can happen that a key is already present. Duplicate keys are not allowed. What happens in case of a duplicate key depends on the value of the `duplicates` option:

Value	Description
`reject`	When a duplicate key is detected, error `XC0106` (pg. 92) is raised
`use-first` (default)	When a duplicate key is detected, the already present entry is used and the duplicate one is discarded.
`use-last`	When a duplicate key is detected, the already present entry is discarded and the duplicate one is used.
`use-any`	When a duplicate key is detected, it is implementation-defined which one is retained.
`combine`	When a duplicate key is detected, the values for both keys are turned into a sequence. Watch out: A sequence of data as value in a map is perfectly fine in the general XPath data model. This means that you can use a result with combined values in your pipeline without problems. However, it is not allowed in JSON, so when you try to serialize such a map as a JSON document, an error will occur.

See Handling duplicates (pg. 91) for an example.

Examples

Basic usage

The following pipeline produces a sequence of 3 documents on the `source` port of `p:json-merge`: An XML, a JSON map and a text document. This is merrily joined into a single map. Notice that the key/value pairs of the JSON map source document are now part of the resulting map.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline>
      <some-xml a="b"/>
    </p:inline>
    <p:inline content-type="application/json" expand-text="false">{"key": 12345, "debug": true}</p:inline>
    <p:inline content-type="text/plain">Hello there!</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-merge/>

</p:declare-step>
```

Result document:

```
{"_1":"\n      <some-xml a=\"b\"\/>\n    ","key":12345,"debug":true,"_3":"Hello there!"}
```

Computing a different key

The following pipeline produces a sequence of 3 documents on the `source` port of `p:json-merge`: An XML document, a JSON map and another XML document. With the `key` option we set the key for the result map to the local name of the root elements: `local-name(/*)`. Notice that what happens with the keys for the second source document, which is itself a map, is not affected.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline>
      <some-xml a="b"/>
    </p:inline>
    <p:inline content-type="application/json" expand-text="false">{"key": 12345, "debug": true}</p:inline>
    <p:inline>
      <some-more-xml c="d"/>
    </p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-merge key="local-name(/*)"/>

</p:declare-step>
```

Result document:

```
{"some-xml":"\n      <some-xml a=\"b\"\/>\n    ","key":12345,"debug":true,"some-more-xml":"\n      <some-
more-xml c=\"d\"\/>\n    "}
```

Handling duplicates

The following pipeline produces a sequence of 2 JSON map documents on the `source` port of `p:json-merge`. Both maps contain an entry with the key `dupkey`. With `duplicates="use-last"` we specify that the last one must be used.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:inline content-type="application/json" expand-text="false">{"dupkey": "a", "debug": true}</p:inline>
    <p:inline content-type="application/json" expand-text="false">{"dupkey": "b"}</p:inline>
  </p:input>
  <p:output port="result"/>

  <p:json-merge duplicates="use-last"/>

</p:declare-step>
```

Result document:

```
{"debug":true,"dupkey":"b"}
```

Additional details

- No document-properties of the source document(s) survive.
- The resulting document has no `base-uri` property.
- If there are no documents appearing on the `source` port, the `result` port returns the empty sequence.

Errors raised

Error code	Description
`XC0106` (pg. 218)	It is a dynamic error if duplicate keys are encountered and option `duplicates` has value "`reject`".
`XC0107` (pg. 218)	It is a dynamic error if a document of a not supported document type appears on port `source` of `p:json-merge`.
`XC0110` (pg. 218)	It is a dynamic error if the evaluation of the XPath expression in option `key` for a given item returns either a sequence, an array, a map, or a function.

2.30 p:label-elements

Labels elements by adding an attribute.

Summary

```
<p:declare-step type="p:label-elements">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="true"/>
  <option name="attribute" as="xs:QName" required="false" select="'xml:id'"/>
  <option name="label" as="xs:string" required="false" select="'concat("_",$p:index)'"/>
  <option name="match" as="xs:string" required="false" select="'*'"/>
  <option name="replace" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:label-elements` step generates a label for each matched element and stores that label in the specified attribute.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to label.
`result`	`output`	`true`	`xml` `html`	`true`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`attribute`	`xs:QName`	`false`	`xml:id`	The name of the attribute that contains the label
`label`	`xs:string` (XPath expression)	`false`	`concat("_", $p:index)`	An XPath expression that computes the value for the label. This expression is evaluated with a matched element as context item. A variable `$p:index` is available that holds the index (sequence number) of the match.
`match`	`xs:string` (XSLT selection pattern)	`false`	*	An XSLT match expression that matches the elements to label.
`replace`	`xs:boolean`	`false`	`true`	Whether to replace existing attributes. If this value is `false`, existing attributes with the same name as mentioned in the `attribute` option are left in peace. If `true` (default), they are replaced.

Description

The `p:label-elements` step performs the following actions:

- It takes the document appearing on its `source` port and finds all the elements matched by the expression in the `match` option.
- For every matched element, it evaluates the expression in the `label` option. This is done with the matched element as context item (so accessible using the dot `.` operator). An additional variable `$p:index` is available that holds the index (sequence number) of the match.
- If the `replace` option is `true` (default), an attribute is added/replaced on the matched element. The name of this attribute is in the `attribute` option. Its value comes from the evaluation of the expression in the `label` option.

  If the `replace` is `false`, an existing attribute with the same name is not replaced.
- After all matches are handled, the resulting document appears on the `result` port.

Examples

Basic usage

The following example uses all the default values of the options of `p:label-elements`. This means an attribute called `xml:id` is added to every element. Values become an underscore followed by the index of the element. Existing `xml:id` attributes are replaced.

Source document:

```
<movies>
   <movie title="Apocalypse now"/>
   <movie title="Dune" xml:id="1234"/>
</movies>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:label-elements/>

</p:declare-step>
```

Result document:

```
<movies xml:id="_1">
   <movie title="Apocalypse now" xml:id="_2"/>
   <movie title="Dune" xml:id="_3"/>
</movies>
```

To make this a little more interesting, let's label the `<movie>` elements only and compute their label based on a generated identifier (using `generate-id()` (https://www.w3.org/TR/xpath-functions-31/#func-generate-id)) and the movie's name (replacing whitespace using `replace()` (https://www.w3.org/TR/xpath-functions-31/#func-replace)). We keep existing `xml:id` attributes. The source document is the same as in the previous example.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:label-elements match="movie" label="generate-
id() || '_' || replace(@title, '\s', '-')" replace="false"/>

</p:declare-step>
```

Result document:

```
<movies>
   <movie title="Apocalypse now" xml:id="id638471686_Apocalypse-now"/>
   <movie title="Dune" xml:id="1234"/>
</movies>
```

Additional details

- `p:label-elements` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.

2.31 p:load

Loads a document.

Summary

```
<p:declare-step type="p:load">
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="content-type" as="xs:string?" required="false" select="()"/>
  <option name="document-properties" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:load` loads a document indicated by a URI and returns this on its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`any`	`false`	The loaded document.

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI for loading the document. In most cases, `p:load` will be used to load a file from disk. An absolute URI for this must start with `file://`. For instance, on Windows, `file:///C:/some/path/document.xml` (although Windows uses backslashes (\) to separate path components, slashes (/) work fine and are more universal). Using a single slash after `file:` also works: `file:/C:/some/path/document.xml`. If this value is relative, it is resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:load` step is stored).
`content-type`	`xs:string?`	`false`	`()`	The content-type of the document to load, for instance `text/plain` or `application/json`. The document is interpreted according to this. If this option is not present, the content-type is determined as described in "Determining the content-type" on page 95.
`document-properties`	`map(xs:QName, item()*)?`	`false`	`()`	Any document-properties for the loaded document.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the loading of the document. Some keys and values are determined by the type of document loaded (see below). Any additional parameters are implementation-defined and therefore dependent on the XProc processor used.

Description

The `p:load` step is one of the few that has no `source` port. It is used to load some document from disk, the web or elsewhere, and returns this document on its `result` port. XProc must know what kind of document it is loading, the mechanism for this is described in "Determining the content-type" on page 95. It is also possible to set document-properties.

What exactly happens depends on the loaded document's content-type:

- For an XML document-type, the document is loaded and interpreted (de-serialized) as XML.

  There is one pre-defined parameter for the `parameters` option: `dtd-validate` (`xs:boolean`). If `true`, DTD validation must be performed when parsing the document.
- Text document-types are loaded "as-is".
- For a JSON document-type, the document is loaded and interpreted (de-serialized) as JSON.

  The `parameters` option recognizes the parsing options as defined for the XPath `parse-json()` (https://www.w3.org/TR/xpath-functions-31/#func-parse-json) function (the `$options` argument).
- For an HTML document-type, the document is loaded and parsed into well-formed XML, even although HTML documents do not have to be well-formed. How this is done exactly is implementation-defined and therefore dependent on the XProc processor used.
- For any other document-type, the document is loaded as a binary document.

There are many ways to load a document into an XProc pipeline. For instance, you could use the `href` attribute of `<p:with-input>`, or use its `<p:document>` child element. The `<p:document>` element is even *defined* as having the same functionality as `p:load`, so there's no difference in functionality.

Why then `p:load`? Its main raison d'être is probably as left-over from the XProc 1.0 days. Using a `p:load` in XProc 1.0 was the only way to dynamically load a document, for instance when you had computed its filename. In recent versions, using AVTs, this is no longer a problem: `<p:with-input href="{$filename}"/>`.

The main reason for using `p:load` probably comes from software engineering: it makes it very explicit in your code what you're doing, an explicit `p:load` stands out more than a nested `<p:document>`. Whether this is reason enough is up to you.

Determining the content-type

When a document is loaded, `p:load` must know its content-type. This is determined as follows:

- When a `content-type` option is specified, this is used.
- If a protocol is used that specifies/returns a content-type, this is used. This is for instance the case when loading documents over HTTP(S).
- If no explicit type information was found, determining the content-type is implementation-defined and therefore dependent on the XProc processor used.

  When loading a document from disk (using the `file://` protocol), in most cases, the XProc processor determines the content-type based on the filename extension. So a `.xml` file will become XML, `.txt` text, etc. What extensions are mapped to what content-type is, again, implementation-defined. However, you can be reasonably sure the most common extensions are interpreted correctly.

Examples

Basic usage

Assume there is an XML document (in the same location as the pipeline) called `extra.xml` with the following contents:

```
<extras>
  <extra>This is nice!</extra>
</extras>
```

The most simple pipeline that uses `p:load` to load this document is:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:load href="extra.xml"/>

</p:declare-step>
```

Result document:

```
<extras>
   <extra>This is nice!</extra>
</extras>
```

Additional details

- With regard to the document-properties of the loaded document:
  - The `content-type` document-property is the content-type of the loaded document. See also "Determining the content-type" on page 95.
  - The `base-uri` document-property is, in most cases, the URI the document is loaded from, as indicated by the `href` option.

    However, the `document-properties` option might also contain a `base-uri` entry. If so, the value in the `document-properties` option is used.
- A content-type can be specified using the `content-type` option and as en entry in the `document-properties` option map. If both are present they must be the same. If not, error `XD0062` (pg. 96) is raised.

Errors raised

Error code	Description
`XD0011` (pg. 220)	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
`XD0023` (pg. 220)	It is a dynamic error if a DTD validation is performed and either the document is not valid or no DTD is found.
`XD0043` (pg. 220)	It is a dynamic error if the `dtd-validate` parameter is `true` and the processor does not support DTD validation.
`XD0049` (pg. 220)	It is a dynamic error if the text value is not a well-formed XML document
`XD0057` (pg. 220)	It is a dynamic error if the text document does not conform to the JSON grammar, unless the parameter liberal is true and the processor chooses to accept the deviation.
`XD0058` (pg. 220)	It is a dynamic error if the parameter duplicates is reject and the text document contains a JSON object with duplicate keys.
`XD0059` (pg. 220)	It is a dynamic error if the parameter map contains an entry whose key is defined in the specification of `fn:parse-json` and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter `escape` with `true()` is also present.
`XD0060` (pg. 220)	It is a dynamic error if the text document can not be converted into the XPath data model
`XD0062` (pg. 220)	It is a dynamic error if the `@content-type` is specified and the document-properties has a "`content-type`" that is not the same.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .
`XD0078` (pg. 220)	It is a dynamic error if the loaded document cannot be represented as an HTML document in the XPath data model.
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.32 p:make-absolute-uris

Make URIs in the document absolute.

Summary

```
<p:declare-step type="p:make-absolute-uris">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="match" as="xs:string" required="true"/>
  <option name="base-uri" as="xs:anyURI?" required="false" select="()"/>
</p:declare-step>
```

The `p:make-absolute-uris` step makes element and/or attribute values in the document appearing on the `source` port absolute by applying a base URI.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to resolve the URIs in.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`match`	`xs:string` (XSLT selection pattern)	`true`		The XSLT match pattern for the attributes and/or elements that hold the URIs to change.
`base-uri`	`xs:anyURI?`	`false`	`()`	The base URI to use for making the matched attributes/elements absolute. If this option is not specified, the base URI of the matched element/attribute in the source document is used. In most cases this will be the location (path on disk) where it originated from.

Description

The `p:make-absolute-uris` step takes the document appearing on its `source` port and searches for elements and/or attributes as indicated by the `match` option. The values of these elements/attributes are taken as URIs and, if relative, resolved against a base URI as specified in the `base-uri` option. If there's no `base-uri` option, the base URI of the element/attribute is used.

Why is this useful? URIs in documents that are edited or received from the web are often relative. They point to other documents, for instance images, that are elsewhere on disk or the web, in a location *relative* to the source document. For instance: `images/picture.jpg`. This allows for flexibility in the location of the documents. However, when processing these URIs and access the referenced documents, the code needs to know how to resolve them into absolute ones. It is practical to arrange all this URI resolving in advance, and this is what `p:make-absolute-uris` is for.

There is an important thing to keep in mind when supplying a value for the `base-uri` option:

- When its value ends with a slash (`/`), it points to a *location*. All URIs are resolved against that location. For instance: `file:///myapp/images/`
- When its value does *not* end with a slash, it points to a document. All URIs are resolved against the location of this document. For instance, for a `base-uri` value `file:///myapp/images/logo.svg`, the URIs are resolved against the location `file:///myapp/images/`

The Basic usage (pg. 98) example shows this difference.

Examples

Basic usage

This example shows what happens to different kinds of URIs when resolved. The `base-uri` value here points to a *location* (because it ends with a `/`).

Source document:

```
<URIs>
   <URI>image.jpg</URI>
   <URI>A/B/C/</URI>
   <URI>/image.jpg</URI>
   <URI>https://xprocref.org/index.html</URI>
</URIs>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:make-absolute-uris match="URI" base-uri="file:///X/Y/Z/"/>

</p:declare-step>
```

Result document:

```
<URIs>
   <URI>file:/…/…/image.jpg</URI>
   <URI>file:/…/…/file:/X/Y/Z/A/B/C/</URI>
   <URI>file:/…/…/image.jpg</URI>
   <URI>https://xprocref.org/index.html</URI>
</URIs>
```

Just to show you the difference, this is what happens when you omit the final `/` from the `base-uri` option. Its value, `file:///X/Y/Z`, now points to a *document* called `Z`. The URIs are resolved against the location of this document.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:make-absolute-uris match="URI" base-uri="file:///X/Y/Z"/>

</p:declare-step>
```

Result document:

```
<URIs>
   <URI>file:/…/…/image.jpg</URI>
   <URI>file:/…/…/file:/X/Y/A/B/C/</URI>
   <URI>file:/…/…/image.jpg</URI>
   <URI>https://xprocref.org/index.html</URI>
</URIs>
```

Additional details

- `p:make-absolute-uris` preserves all document-properties of the document(s) appearing on its `source` port.
- The `match` option must match attributes or elements. If anything else is matched, error `XC0023` (pg. 99) is raised.
- A relative value for the `base-uri` option is resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:make-absolute-uris` is stored). This is very probably not what you want.
- If no `base-uri` option is specified and an element/attribute matched has no base URI also, the result is implementation-defined and therefore dependent on the XProc processor used.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

2.33 p:markdown-to-html

Converts a Markdown document into HTML.

Summary

```
<p:declare-step type="p:markdown-to-html">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="html" sequence="false"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:markdown-to-html` step converts a Markdown (https://www.markdownguide.org/) document appearing on its `source` port into HTML. The result appears on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The Markdown (https://www.markdownguide.org/) document to transform.
`result`	`output`	`true`	`html`	`false`	The resulting HTML document.

Options:

Name	Type	Req?	Default	Description
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the conversion. The XProc specification does not define any parameters for this option. A specific XProc processor (or renderer used) might define its own.

Description

The `p:markdown-to-html` step converts a Markdown (https://www.markdownguide.org/) document appearing on its `source` port into HTML. There are several flavors of Markdown, for instance CommonMark (https://spec.commonmark.org/0.31.2/). Which Markdown flavors are supported by `p:markdown-to-html` is implementation-defined and therefore dependent on the XProc processor used. The resulting HTML appears on the `result` port.

Examples

Basic usage

Assume we have a Markdown document that looks like this:

```
# Example Markdown document

This is an example of a Markdown document to show what the conversion to HTML by `p:markdown-to-
html` looks like.
```

We can convert this into HTL using `p:markdown-to-html`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:markdown-to-html/>

</p:declare-step>
```

Result document:

```
<html xmlns="http://www.w3.org/1999/xhtml">
   <head/>
   <body>
      <h1>Example Markdown document</h1>
      <p>This is an example of a Markdown document to show what the conversion to HTML by <code>p:markdown-
to-html</code> looks like.</p>
   </body>
</html>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

2.34 p:message

Produces a message.

Summary

```
<p:declare-step type="p:message">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <option name="select" as="item()*" required="true"/>
  <option name="test" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:message` step produces a message that is, usually, printed on the console. The effect (when the `test` option is `true`) is the same as using a `message`/`p:message` attribute on a step.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The source document(s)
`result`	`output`	`true`	`any`	`true`	The resulting document(s). These will be exactly the same as what appeared on the `source` port.

Options:

Name	Type	Req?	Default	Description
`select`	`item()*` (XPath expression)	`true`		The message to produce
`test`	`xs:boolean`	`false`	`true`	If `true`, the message in the `select` attribute is produced.

Description

Steps in general can produce messages by using the `message` (for steps in the XProc namespace) or `p:message` (for steps in other namespaces) attribute. What "produce" here actually means is implementation-defined and therefore depends on the XProc processor used. However, usually it means "printed on the console" and/or "output through `stdout`": a command-line message appears when the processor executes the step.

The `p:message` step is an alternative way to produce these messages. When the `test` option is `true` (default), the expression in the `select` option is evaluated and the result is produced/shown, as a message. If the `test` option is `false`, nothing happens.

The step itself, irrespective of the value of the `test` option, simply passes what it gets on its `source` port unaltered to its `result` port. In other words, it acts as a `p:identity` (pg. 78) step.

Examples

Basic usage

Assume you have a pipeline that does some preliminary things (getting documents, computing variables, etc.) and then starts the real computation of something. In-between you want a message that says the computation has started, but only when enabled by an option. Here is an example of how to do this using the `p:message` step:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" xmlns:xs="http://www.w3.org/2001/
XMLSchema" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:option name="debug-messages-on" as="xs:boolean" select="true()"/>

  <!-- Some preliminary stuff… -->

  <p:message test="{$debug-messages-on}" select="'Starting computation at ' || current-dateTime()"/>

  <!-- Steps that implement the computation… -->

</p:declare-step>
```

It is certainly possible to implement this without the `p:message` step, using a `p:identity` (pg. 78) step with a `message` attribute:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" xmlns:xs="http://www.w3.org/2001/
XMLSchema" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:option static="true" name="debug-messages-on" as="xs:boolean" select="true()"/>

  <!-- Some preliminary stuff… -->

  <p:identity use-when="$debug-messages-on" message="Starting computation at {current-dateTime()}"/>

  <!-- Steps that implement the computation… -->

</p:declare-step>
```

Please notice the differences between the two examples:

- In the first example the on/off switch for the message, here the `debug-messages-on` option, is *dynamic*. It can be computed/set during run-time, if necessary.

  However, in the second example, this on/off switch is referenced in a `use-when` attribute. All `use-when` attributes are evaluated during compile-time, and therefore the `debug-messages-on` option must be *static* (hence its `static="true"` attribute). The only time you can change/set this option, and turn messages on/off, is when invoking the pipeline.
- The `select` option of the `p:message` step is an XPath expression. The value of the `message` attribute is an AVT (Attribute-Value Template). This results in a very different syntax, while the result is identical.

Additional details

- `p:message` preserves all document-properties of the document(s) appearing on its `source` port.

2.35 p:namespace-delete

Deletes namespaces from a document.

Summary

```
<p:declare-step type="p:namespace-delete">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="prefixes" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:namespace-delete` step deletes namespaces, for which the prefixes are listed in the `prefixes` option, from elements and attributes in the document appearing on its `source`. The resulting document appears on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to delete the namespaces from.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document

Options:

Name	Type	Req?	Description
`prefixes`	`xs:string`	`true`	A whitespace-separated list of namespace-prefixes. These prefixes must be defined in your pipeline. The namespaces associated with the prefixes are removed.

Description

The `p:namespace-delete` step takes the value of its `prefixes` option, which must be a whitespace separated list of namespace-prefixes, and finds out which namespaces are associated with these prefixes. These namespace prefixes must be defined in the pipeline. It then uses this list to delete these namespaces from elements and attributes in the document appearing on the `source` port. Elements and attributes that were in one of these namespaces are now in the no-namespace. The resulting document appears on the `result` port.

Note that matching is done on namespace *name*, not on *namespace-prefix*. This means that the prefix as used in the `prefixes` option might be different from the one used in the document to delete the namespace from. See the Basic usage with different namespace-prefixes (pg. 102) example.

Examples

Basic usage

The following example deletes the `#myconfig` namespace, associated with the prefix `con`, from the source document.

Source document:

```
<config xmlns:con="#myconfig" con:status="special">
   <con:thing>button</con:thing>
</config>
```

Pipeline document:

```
<p:declare-step xmlns:con="#myconfig" xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-delete prefixes="con"/>

</p:declare-step>
```

Result document:

```
<config status="special">
   <thing>button</thing>
</config>
```

Basic usage with different namespace-prefixes

The following example again deletes the `#myconfig` namespace. However, the namespace-prefix used in the pipeline is different from the one used in the source document.

Source document:

```
<config xmlns:con="#myconfig" con:status="special">
   <con:thing>button</con:thing>
</config>
```

Pipeline document:

```
<p:declare-step xmlns:ns1="#myconfig" xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-delete prefixes="ns1"/>

</p:declare-step>
```

Result document:

```
<config status="special">
   <thing>button</thing>
</config>
```

Additional details

- `p:namespace-delete` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0108` (pg. 218)	It is a dynamic error if any prefix is not in-scope at the point where the `p:namespace-delete` occurs.
`XC0109` (pg. 218)	It is a dynamic error if a namespace is to be removed from an attribute and the element already has an attribute with the resulting name. For instance, removing the namespace with the `ns1` prefix will raise this error when applied to `<something ns1:status="ok" status="bad"/>`.

2.36 p:namespace-rename

Renames a namespace to a new URI.

Summary

```
<p:declare-step type="p:namespace-rename">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="apply-
to" as="item()*" required="false" select="'all'" values="('all','elements','attributes')"/>
  <option name="from" as="xs:anyURI?" required="false" select="()"/>
  <option name="to" as="xs:anyURI?" required="false" select="()"/>
</p:declare-step>
```

The `p:namespace-rename` step renames any namespace declaration or use of a namespace in a document to a new value.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to rename the namespace in.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`apply-to`	`item()*`	`false`	`all`	Whether to apply the changes to elements, attributes or both. See "The apply-to option" on page 104 below.
`from`	`xs:anyURI?`	`false`	`()`	The namespace URI to rename from.
`to`	`xs:anyURI?`	`false`	`()`	The namespace URI to rename to.

Description

The `p:namespace-rename` step changes a namespace in a document into another namespace. It affects both the namespace bindings and the namespace usage. Usually that's all there is to it, you can rely on the step to take care of all the details (for those that need to know, see "Detailed processing" on page 104).

If you want to remove a namespace altogether, you need `p:namespace-delete` (pg. 101).

The apply-to option

The `apply-to` option can take the following values:

Value	Description
`all` (default)	Apply the changes to both elements and attributes.
`attributes`	Apply the changes to attributes only.
`elements`	Apply the changes to elements only.

The main reason for having an `apply-to` option is to avoid renaming attributes when the `from` option specifies no namespace. This happens when you want to turn a document that is not in a namespace into some namespace. Often however, attributes are never in a namespace. By setting the `apply-to` option to `elements`, the attributes are not affected.

Detailed processing

The step takes the document appearing on its `source` port and examines it for occurrences of the namespace mentioned in the `from` option:

- A *namespace binding* with the `from` value, either defining a prefix (`xmlns:…="…"`) or a default namespace (`xmlns="…"`) declaration, gets the value as specified in the `to` option.

  If the `from` option is absent or the empty string, no bindings are changed/removed.

  If the `to` option is not specified or the empty string, the binding is removed.
- Depending on the value of the `apply-to` option (see "The apply-to option" on page 104), elements and/or attributes that are in the `from` namespace are turned into the `to` namespace.

  If the `from` option is absent to the empty string, the changes apply to elements/attributes without a namespace (that are in the no-namespace).

  If the `to` option is not specified or the empty string, the namespace of the element/attribute is removed (putting it into the no-namespace).

Examples

Basic usage

Source document:

```
<some-document xmlns="#some-namespace">
   <contents a="b"/>
</some-document>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-rename from="#some-namespace" to="#some-other-namespace"/>

</p:declare-step>
```

Result document:

```
<some-document xmlns="#some-other-namespace">
   <contents a="b"/>
</some-document>
```

This also works when the source document uses a namespace prefix:

```
<ns:some-document xmlns:ns="#some-namespace">
   <ns:contents a="b"/>
</ns:some-document>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-rename from="#some-namespace" to="#some-other-namespace"/>

</p:declare-step>
```

Result document:

```
<ns:some-document xmlns:ns="#some-other-namespace">
   <ns:contents a="b"/>
</ns:some-document>
```

Renaming to a namespace

If you rename a document from the no-namespace into a namespace, you usually *don't* want to rename the attributes to that namespace as well. However, if you don't do anything special, this is exactly what will happen:

Source document:

```
<some-document>
   <contents a="b"/>
</some-document>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-rename to="#some-namespace"/>

</p:declare-step>
```

Result document:

```
<some-document xmlns="#some-namespace">
   <contents xmlns:_1="#some-namespace" _1:a="b"/>
</some-document>
```

The XProc processor invents a namespace prefix, and uses this to put the attribute(s) in the target namespace as well. To avoid this, set the `apply-to` option to `elements:`

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:namespace-rename to="#some-namespace" apply-to="elements"/>

</p:declare-step>
```

Result document:

```
<some-document xmlns="#some-namespace">
   <contents a="b"/>
</some-document>
```

Additional details

- `p:namespace-rename` preserves all document-properties of the document(s) appearing on its `source` port.
- If the value of the `from` and `to` option are the same nothing happens. The step acts like a `p:identity` (pg. 78) step.

Errors raised

Error code	Description
`XC0014` (pg. 216)	It is a dynamic error if the XML namespace (`http://www.w3.org/XML/1998/namespace`) or the XMLNS namespace (`http://www.w3.org/2000/xmlns/`) is the value of either the `from` option or the `to` option.
`XC0092` (pg. 217)	It is a dynamic error if as a consequence of changing or removing the namespace of an attribute the attribute's name is not unique on the respective element.

2.37 p:os-exec

Runs an external command.

Summary

```
<p:declare-step type="p:os-exec">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <output port="error" primary="false" content-types="any" sequence="true"/>
  <output port="exit-status" primary="false" content-types="application/xml" sequence="false"/>
  <option name="command" as="xs:string" required="true"/>
  <option name="args" as="xs:string*" required="false" select="()"/>
  <option name="cwd" as="xs:string?" required="false" select="()"/>
  <option name="error-content-type" as="xs:string" required="false" select="'text/plain'"/>
  <option name="failure-threshold" as="xs:integer?" required="false" select="()"/>
  <option name="path-separator" as="xs:string?" required="false" select="()"/>
  <option name="result-content-type" as="xs:string" required="false" select="'text/plain'"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:os-exec` step runs the external command as specified in the `command` and `args` options. It passes the input that arrives on its `source` port as standard input to the command. The standard output and standard error of the command are returned on the `result` and `error` ports.

Ports:

Port	Type	Primary?	Content types	Seq?	Description

`source`	`input`	`true`	`any`	`true`	The document that appears on the standard input of the external command. If the `source` port is empty, the command receives nothing on standard input.
					This document is serialized (as if written to disk) first. For this, the value of the `serialization` document-property and that of the `serialization` option are used. See the `serialization` option for more information.
					The port accepts zero or one document. For multiple documents, error `XC0032` (pg. 109) is raised.
					If you want the source port to be empty, you must specify this:
					`<p:with-input port="source">`
					`  <p:empty/>`
					`</p:with-input>`

`result`	`output`	`true`	`any`	`true`	The result of the external command: what was written by the command on the standard output. The standard output of the command is processed as if it was read from disk by `p:load` (pg. 94). The value of the `result-content-type` option of `p:os-exec` is taken as the value for the `content-type` option of `p:load` (pg. 94). If there is no content on the standard output, this port will be empty.
`error`	`output`	`false`	`any`	`true`	The error result of the external command: what was written by the command on the standard error. The standard error of the command is processed as if it was read from disk by `p:load` (pg. 94). The value of the `error-content-type` option of `p:os-exec` is taken as the value for the `content-type` option of `p:load` (pg. 94). If there is no content on the standard error, this port will be empty.
`exit-status`	`output`	`false`	`application/xml`	`false`	A `<c:result>` element containing the system exit status, as an integer (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Options:

Name	Type	Req?	Default	Description
`command`	`xs:string`	`true`		The external command to execute (without arguments).
`args`	`xs:string*`	`false`	`()`	The arguments for the external command (as a sequence of strings).
`cwd`	`xs:string?`	`false`	`()`	The current working directory for the execution of the command. If this option is left unspecified, the current working directory will be implementation-defined and therefore depends on the XProc processor used. To resolve variations in the syntax of directory specifications, the value supplied for this option will be normalized using the `p:urify() (https://spec.xproc.org/master/head/xproc/#f.urify)` function.
`error-content-type`	`xs:string`	`false`	`text/plain`	The content type of the command's error result (its standard error). See the description of the `error` port.
`failure-threshold`	`xs:integer?`	`false`	`()`	If a value for this option is supplied and the command exit status is greater than this value, error `XC0064` (pg. 109) is raised.
`path-separator`	`xs:string?`	`false`	`()`	If specified, every occurrence of this character that occurs in the `command`, `args`, or `cwd` options will be replaced by the platform-specific path separator character. The value for this option must be exactly one character long. if not, error `XC0063` (pg. 109) is raised.
`result-content-type`	`xs:string`	`false`	`text/plain`	The content type of the command's result (its standard output). See the description of the `result` port.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/) for serializing the document on the `source` port, before it is passed as the standard input of the command. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence). Example: `serialization="map{'method': 'text'}"`

Description

The `p:os-exec` step can be used to run an external command. For instance, a Python script or post-process some result with something not available in XProc.

The command itself is specified using the `command` and `args` options. Every string in the `args` option is a separate argument (also when it contains spaces). What appears on the `result` port is passed as the command's standard input.

The command's output and error information is returned on the `result` and `error` ports. You can use the `result-content-type` and `error-content-type` options to tailor how this comes out.

Examples

Basic usage

This example runs the (Windows) `dir` command (that shows a directory overview) on the `data/` subdirectory of where the pipeline is stored. It does so by starting the Windows command processor (called `cmd`) with the arguments `/C dir data`. This is the same as typing `dir data` on the Windows command line.

A problem we have here is that we need to set the current working directory (in the `cwd`) to the location of the pipeline. The example computes this, based on the `static-base-uri()` of the pipeline, using regular expression magic. The `cwd` option of `p:os-exec` does not accept a URI, so we have to strip the `file:/` in front also. The result is stored in the `$cwd` variable.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:variable name="cwd" select="static-base-uri() =&gt; replace('^file:/+', '') =&gt; replace('[^/\\]+
$', '')"/>

  <p:os-exec command="cmd" cwd="{$cwd}">
    <p:with-option name="args" select="('/C', 'dir', 'data')"/>
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:os-exec>

</p:declare-step>
```

Result document (text):

```
Volume in drive C is OS
 Volume Serial Number is 9EA2-E853

 Directory of C:\…\data

03/06/2025  13:42    <DIR>          .
03/06/2025  14:33    <DIR>          ..
05/02/2025  14:05                46 x1.xml
05/02/2025  14:05                46 x2.xml
               2 File(s)             92 bytes
               2 Dir(s)  512.762.478.592 bytes free
```

Additional details

- The documents appearing on the output ports only have a `content-type` property. They have no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0032` (pg. 216)	It is a dynamic error if more than one document appears on the `source` port of the `<p:os-exec>` step.
`XC0033` (pg. 216)	It is a dynamic error if the command cannot be run.
`XC0034` (pg. 216)	It is a dynamic error if the current working directory cannot be changed to the value of the `cwd` option.
`XC0063` (pg. 216)	It is a dynamic error if the `path-separator` option is specified and is not exactly one character long.
`XC0064` (pg. 217)	It is a dynamic error if the exit code from the command is greater than the specified `failure-threshold` value.

2.38 p:os-info

Returns information about the operating system.

Summary

```
<p:declare-step type="p:os-info">
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
</p:declare-step>
```

The `p:os-info` step returns information about the operating system the XProc processor is running on, which appears on the `result` port as an XML document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/xml`	`false`	A `<c:result>` element describing properties of the operating system. See "The result document" on page 110 for more information.

Description

The `p:os-info` step gathers information about the operating system the XProc processor is running on. This results in an XML document on the `result` port.

The result document

The document appearing on the `result` port has a `<c:result>` root element (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace).

Basic information is contained in a number of mandatory attributes. XProc processors may add other attributes with operating system information, but these are, of course, implementation-defined and therefore depend on the XProc processor used. Attribute values can be the empty string if they do not apply to the runtime environment (which will rarely happen).

Environment variables are listed in `<c:environment>` child elements.

```
<c:result cwd
          file-separator
          os-architecture
          os-name
          os-version
          path-separator
          user-home
          user-name
          *? >
  <c:environment name="…" value="…">*
</c:result>
```

Attribute	#	Description
`cwd`	1	The current working directory (in operating system dependent notation). For instance `C:\user\erik\data\bin`.
`file-separator`	1	The file/path separator character. For instance `/` on Unix or `\` on Windows.
`os-architecture`	1	The operating system architecture. For instance `i386`.
`os-name`	1	The name of the operating system. For instance `Mac OS X` or `Windows 10`.
`os-version`	1	The version of the operating system. For instance `10.0`.
`path-separator`	1	The path separator character. For instance `:` on Unix or `;` on Windows.
`user-home`	1	The home directory of the current user (in operating system dependent notation). For instance `C:\user\erik\data`.
`user-name`	1	The (login) name of the current user. For instance `erik`
*	?	The XProc processor may add other attributes with information about the operating system (in a different namespace). These attributes, their values and meaning are implementation-defined and therefore depend on the XProc processor used.

Child element	#	Description
`c:environment name="…" value="…"`	*	Name and value of an environment variable.

Examples

Basic usage

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:os-info/>

</p:declare-step>
```

On the system used to develop the XProcRef site on, the *partial* result is:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step"
          os-name="Windows 11"
          user-home="C:\Users\erik"
          file-separator="\"
          user-name="erik"
          path-separator=";"
          os-version="10.0"
          cwd="C:\…"
          os-architecture="amd64">
   <c:environment name="…" value="…"/>
</c:result>
```

Some information (especially the names and values of the environment variables) is left out for privacy reasons.

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

2.39 p:pack

Merges two document sequences, pair-wise.

Summary

```
<p:declare-step type="p:pack">
  <input port="source" primary="true" content-types="text xml html" sequence="true"/>
  <output port="result" primary="true" content-types="application/xml" sequence="true"/>
  <input port="alternate" primary="false" content-types="text xml html" sequence="true"/>
  <option name="wrapper" as="xs:QName" required="true"/>
  <option name="attributes" as="map(xs:QName, xs:anyAtomicType)?" required="false" select="()"/>
</p:declare-step>
```

The `p:pack` step takes the document sequences appearing on its `source` and `alternate` ports and combines these documents in a pair-wise fashion, wrapping the pairs in a wrapper element.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text xml html`	`true`	The first document sequence to merge.
`result`	`output`	`true`	`application/xml`	`true`	The resulting merged document sequences.
`alternate`	`input`	`false`	`text xml html`	`true`	The second document sequence to merge.

Options:

Name	Type	Req?	Default	Description
`wrapper`	`xs:QName`	`true`		The element to wrap the document pairs in.
`attributes`	`map(xs:QName, xs:anyAtomicType)?`	`false`	`()`	An optional map with entries (attribute name, attribute value). The attributes specified in this map are created on the wrapper element. Specifying attributes using this option works the same as performing a `p:pack` step (without an `attributes` option), directly followed by a `p:set-attributes` (pg. 124) step.

Description

The `p:pack` step takes two document sequences, one on its `source` and one on its `alternate` port. It then takes the first document in both sequences, concatenates these and wraps this in a wrapper element as indicated by the `wrapper` option. The same is done for the second pair, etc. The resulting wrapped document pairs are emitted on the `result` port.

If `p:pack` reaches the end of one input sequence before the other, the remaining documents will be wrapped as single documents.

Examples

Basic usage

The following pipeline provides `p:pack` with two document sequences. The pairs are wrapped in a `<pair-wrapper>` element. Since the sequence on the `alternate` port is one document longer than the one on the `source` port, the remaining document `<alternate-doc-3/>` is wrapped as single document.

The resulting document sequence here is wrapped using `p:wrap-sequence` (pg. 187), just to show the results.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:pack wrapper="pair-wrapper">
    <p:with-input port="source">
      <source-doc-1/>
      <source-doc-2/>
    </p:with-input>
    <p:with-input port="alternate">
      <alternate-doc-1/>
      <alternate-doc-2/>
      <alternate-doc-3/>
    </p:with-input>
  </p:pack>

  <p:wrap-sequence wrapper="all-packed-results"/>

</p:declare-step>
```

Result document:

```
<all-packed-results>
   <pair-wrapper>
      <source-doc-1/>
      <alternate-doc-1/>
   </pair-wrapper>
   <pair-wrapper>
      <source-doc-2/>
      <alternate-doc-2/>
   </pair-wrapper>
   <pair-wrapper>
      <alternate-doc-3/>
   </pair-wrapper>
</all-packed-results>
```

Additional details

- No document-properties of the source/alternate documents survive.
- The resulting document(s) have no `base-uri` property.

2.40 p:rename

Renames nodes in a document.

Summary

```
<p:declare-step type="p:rename">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="new-name" as="xs:QName" required="true"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
</p:declare-step>
```

The `p:rename` step renames elements, attributes, or processing-instruction nodes, specified by an XSLT selection pattern, in the document appearing on its `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to rename the nodes in.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`new-name`	`xs:QName`	`true`		The new name for the matched nodes.
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern for the nodes to rename, as a string.

Description

Using `p:rename`, it becomes easy to rename elements, attributes, or processing-instructions in your document. The step takes the XSLT match pattern in the `match` option and holds this against the document appearing on its `source` port. Any matching nodes are renamed to the name provided in the `new-name` option. Matched nodes must be elements, attributes, or processing-instructions (any other match results in error `XC0023` (pg. 115)).

Examples

Basic usage

The following example renames an element, an attribute and a processing-instruction in the source document.

Source document:

```
<things>
   <thing name="screw" id="A123"/>
   <thing name="bolt" id="A789"/>
   <?convert debug="true"?>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:rename match="/*/thing" new-name="Thing"/>
  <p:rename match="@name" new-name="thing-name"/>
  <p:rename match="processing-instruction(convert)" new-name="debug-processing"/>

</p:declare-step>
```

Result document:

```
<things>
   <Thing thing-name="screw" id="A123"/>
   <Thing thing-name="bolt" id="A789"/>
   <?debug-processing debug="true"?>
</things>
```

Renaming to an existing attribute

This example shows that when an attribute is renamed to one that is already present, the existing attribute is deleted.

Source document:

```
<things>
   <thing name="screw" id="A123" thing-name="something else"/>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:rename match="@name" new-name="thing-name"/>

</p:declare-step>
```

Result document:

```
<things>
   <thing thing-name="screw" id="A123"/>
</things>
```

Additional details

- `p:rename` preserves all document-properties of the document(s) appearing on its `source` port.
- If an attribute is renamed to an attribute that already exists on this element, this existing attribute is deleted. See the Renaming to an existing attribute (pg. 114) example.
- If an `xml:base` attribute is renamed to something else, the underlying base URI of the element is *not* changed.

  If an attribute is renamed *to* `xml:base`, the base URI of the underlying element is changed to the value of this attribute.

Errors raised

Error code	Description
`XC0013` (pg. 216)	It is a dynamic error if the pattern matches a processing instruction and the new name has a non-null namespace.
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.

2.41 p:replace

Replace nodes with a document.

Summary

```
<p:declare-step type="p:replace">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <input port="replacement" primary="false" content-types="text xml html" sequence="false"/>
  <option name="match" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:replace` step takes the document appearing on its `source` port and replaces nodes matching the `match` option with the document appearing on the `replacement` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document in which to replace nodes.
`result`	`output`	`true`	`text` `xml` `html`	`false`	The resulting document.
`replacement`	`input`	`false`	`text` `xml` `html`	`false`	The document to replace the nodes with.

Options:

Name	Type	Req?	Description
`match`	`xs:string` (XSLT selection pattern)	`true`	The XSLT match pattern for the nodes to replace, as a string.

Description

The `p:replace` step takes the XSLT match pattern in the `match` option and holds this against the document appearing on its `source` port. Any matching nodes are replaced by the document on the `replacement` port. The resulting document is emitted on the `result` port.

This step replaces matched nodes with a complete document. If you need to replace matched nodes with (just) strings, have a look at the `p:string-replace` (pg. 134) step.

Examples

Basic usage

The following example replaces all `<thing>` elements with `<another-thing/>` elements.

Source document:

```
<things>
   <thing>
      <contents/>
   </thing>
   <thing/>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:replace match="thing">
    <p:with-input port="replacement">
      <another-thing/>
    </p:with-input>
  </p:replace>

</p:declare-step>
```

Result document:

```
<things>
   <another-thing/>
   <another-thing/>
</things>
```

Alternative approach

What the `p:replace` step does is similar to what a simple `p:viewport` instruction does: it takes a matched node and replaces it with something. This pipeline has the same functionality as the one in Basic usage (pg. 115):

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:viewport match="thing">
    <p:identity>
      <p:with-input>
        <another-thing/>
      </p:with-input>
    </p:identity>
  </p:viewport>

</p:declare-step>
```

Result document (using the same input document as in Basic usage (pg. 115)):

```
<things>
   <another-thing/>
   <another-thing/>
</things>
```

Additional details

- For obvious reasons, you cannot replace attributes and namespace nodes.
- Replacing by `p:replace` is not recursive. In other words: there are no replacements in a replacement.
- `p:replace` preserves all document-properties of the document(s) appearing on its `source` port.
  There is one exception: if the resulting document contains only text, the `content-type` document-property is changed to `text/plain` and the `serialization` document-property is removed.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.

2.42 p:run

Runs a dynamically loaded pipeline.

Summary

The `p:run` step executes a dynamically loaded pipeline within the current pipeline.

Description

In using XProc, there are cases where it is useful to be able to run a *dynamically* loaded pipeline. The `p:run` step makes this possible. For instance:

- Assume you have an XProc pipeline that processes/executes some DSL (Domain Specific Language). This language refers to XProc pipelines (by URI) that need to be executed as part of the DSL processing.
- Your pipeline gets so complicated that it becomes easier to dynamically *construct* another pipeline and run this as part of your main pipeline.

Without `p:run`, all this wouldn't be possible. Unfortunately, because various requirements surrounding dynamic pipeline execution, the `p:run` step is *very* different from other steps: it has a totally different set of child elements. Therefore, the step will be explained as if it was a separate XML element:

```
<p:run name? = xs:NCName >
  ( <p:with-input> |
    <run-input>* |
    <run-option>* |
    <output>*  )
</p:run>
```

Attribute	#	Type	Description
`name`	?	`xs:NCName`	The standard XProc step `name` attribute.

Child element	#	Description
`p:with-input`	1	Anonymous input port that receives the pipeline to run. This is *not* the step's primary port! Its attributes and child elements are similar to the standard XProc `<p:with-input>` element, except that, since the port is anonymous, it does not have a `port` attribute.
`run-input`	1	Element for connecting the input ports of the pipeline being run. Its attributes and child elements are similar to the standard XProc `<p:with-input>` element, except that it has an additional boolean `primary` attribute.
`run-option`	1	Element for passing options to the pipeline being run. Its attributes and child elements are similar to the standard XProc `<p:with-option>` element, except that it has an additional boolean `static` attribute.
`output`	1	Element that declares an output port of the pipeline being run.

In handling all this, there are quite a few details involved, which are discussed in the sections below. However, in the vast majority of cases, using `p:run` is not very difficult. You may want to look at the examples before diving into the details.

Details specifying the pipeline to run

The pipeline to run (the dynamically executed pipeline) must be provided on the anonymous input port of `p:run` (its "pipeline" port). This must of course be a valid XProc pipeline and it must have a `version` attribute.

You connect this anonymous "pipeline" port by adding a single `<p:with-input>` child element to the `p:run`, without a `port` attribute. For example, assume there is a step in my pipeline, named `pipeline-generating-step`, that produces a pipeline on its primary `result` port. To execute this pipeline using `p:run`, I would have to write:

```
<p:run>
  <p:with-input pipe="result@pipeline-generating-step"/>
  …
</p:run>
```

Take care: this anonymous "pipeline" port is *not* the primary port of `p:run`. So you must *always connect it*, even if the pipeline to run flows out of the primary output port of the step right before (in XProc terms: even if it is the DRP, the Default Readable Port).

Details connecting input ports

The `<p:run-input>` element is used to connect the input ports of the dynamically executed pipeline. Its attributes and child elements are almost similar to the standard XProc `<p:with-input>` element.

For example, to connect the primary `source` input port of the dynamically executed pipeline to some document on disk, you could write:

```
<p:run>
  <p:with-input href="my-dynamic-pipeline.xpl"/>
  <p:run-input port="source" href="doc.xml"/>
  …
</p:run>
```

There are a few things you need to keep in mind when connecting the input ports of the dynamically executed pipeline:

- The additional boolean `primary` attribute of `<p:run-input>` can be used to declare that this port in the dynamically executed pipeline is primary. Its default value depends:
  - If there is a single `<p:run-input>` element, its default value is `true`.
  - If there are multiple `<p:run-input>` elements, its default value is `false`.
- If the primary input port of the pipeline to run is not explicitly connected to somewhere (in its `<p:run-input>` element), it gets connected to the default readable port of `p:run`.
- The name of the primary input port, as declared by the `<p:run-input>` element of `p:run`, *must* be the same as the name of the primary input port of the dynamically executed step. If not, error `XC0206` (pg. 121) is raised.
- Any input ports of the dynamically executed step for which there is no corresponding `<p:run-input>` element receive an empty connection (`<p:empty/>`).
- Any input ports mentioned in `<p:run-input>` elements that do not exist in the dynamically executed pipeline are silently ignored.

Details specifying options

Options for the dynamically executed pipeline can be specified using one or more `<p:run-option>` elements. Its attributes and child elements are almost similar to the standard XProc `<p:with-option>` element.

There are a few additional details to reckon with:

- The `<p:run-option>` element has an additional boolean attribute `static`, which defaults to `false`. Using `static="true"` allows you to specify static options for the pipeline to run.
- Any options of the dynamically executed pipeline that are not specified using `<p:run-option>` are handled as expected: if they are required, an error will be raised, and if not, their default value applies.
- Any options set by `<p:run-option>` that do not exist in the dynamically executed pipeline are silently ignored.

Details specifying output ports

To be able to access the output ports of the dynamically executed pipeline, you have to declare these ports on the `p:run` invocation using `<p:output>` elements. For example, if your dynamic pipeline has a primary `result` output port and you want to access what comes out of this port in the pipeline that does the `p:run`, you *must* write something like:

```
<p:run>
  <p:with-input href="my-dynamic-pipeline.xpl"/>
  <p:output port="result" primary="true"/>
  …
</p:run>
```

A few details to consider:

- The `<p:output>` child element of `p:run` has the same definition as a normal `<p:output>` element, but here it may not establish a connection to another port/step in the pipeline. In other words: you cannot use the `pipe` attribute or a child `<p:pipe>` element.
- The name of the primary output port, as declared by the `<p:output>` element of `p:run`, *must* be the same as the name of the primary output port of the dynamically executed step. If not, error `XC0207` (pg. 121) is raised.

- If the dynamically executed pipeline has output ports that are not declared in `<p:output>` elements of `p:run`, their outputs will be silently discarded.
- If the `p:run` step declares additional output ports that are not present in the dynamically executed pipeline, their outputs will be empty (`<p:empty/>`).

Examples

Basic usage

Suppose we have some kind of XML content in which we want to dynamically generate other content using XProc pipelines. For example:

```
<body>
  <p>Let's do some additions:</p>
  <ul>
    <li>
      <generate href-pipeline="add-them.xpl" a="1" b="1"/>
    </li>
    <li>
      <generate href-pipeline="add-them.xpl" a="7" b="3"/>
    </li>
  </ul>
</body>
```

The `<generate>` elements invokes another pipeline (named in its `href-pipeline` attribute) using `p:run`:

- The input to these generator pipelines, on their primary `source` port, is the `<generate>` element itself (so it can access any attributes/child elements).
- The output of these generator pipelines, on their primary `result` port, is inserted back into the original document, replacing the `<generate>` element.

For this example, we only use one generator pipeline, called `add-them.xpl`. This simply adds the attributes `a` and `b` and reports about this:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" xmlns:xs="http://www.w3.org/2001/
XMLSchema" version="3.0" exclude-inline-prefixes="#all">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:variable name="a" as="xs:integer" select="xs:integer(/*/@a)"/>
  <p:variable name="b" as="xs:integer" select="xs:integer(/*/@b)"/>

  <p:identity>
    <p:with-input>
      <p>Adding {$a} to {$b} results in {$a + $b}!</p>
    </p:with-input>
  </p:identity>

</p:declare-step>
```

The main pipeline and its output are as follows:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:viewport match="generate">
    <p:run>
      <p:with-input href="{/*/@href-pipeline}"/>
      <p:run-input port="source"/>
      <p:output port="result"/>
    </p:run>
  </p:viewport>

</p:declare-step>
```

Result document:

```
<body>
   <p>Let's do some additions:</p>
   <ul>
      <li>
         <p>Adding 1 to 1 results in 2!</p>
      </li>
      <li>
         <p>Adding 7 to 3 results in 10!</p>
      </li>
   </ul>
</body>
```

Using options

Building on the Basic usage (pg. 119) example, we're going to add and use an option that specifies the language of the generator output. For now this will only accept `nl` for Dutch. Any other language will default to English.

The extended version of the generator pipeline, called `add-them-extended.xpl`, looks like this:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" xmlns:xs="http://www.w3.org/2001/
XMLSchema" version="3.0" exclude-inline-prefixes="#all">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:option name="language" as="xs:string" required="true"/>

  <p:variable name="a" as="xs:integer" select="xs:integer(/*/@a)"/>
  <p:variable name="b" as="xs:integer" select="xs:integer(/*/@b)"/>

  <p:choose>
    <p:when test="$language eq 'nl'">
      <p:identity>
        <p:with-input>
          <p>Als we {$a} optellen bij {$b} krijgen we {$a + $b}!</p>
        </p:with-input>
      </p:identity>
    </p:when>
    <p:otherwise>
      <!-- Default language is English: -->
      <p:identity>
        <p:with-input>
          <p>Adding {$a} to {$b} results in {$a + $b}!</p>
        </p:with-input>
      </p:identity>
    </p:otherwise>
  </p:choose>

</p:declare-step>
```

Let's generate a Dutch version of our output:

Source document:

```
<body>
   <p>Laten we optellen:</p>
   <ul>
      <li>
         <generate href-pipeline="add-them-extended.xpl" a="1" b="1"/>
      </li>
      <li>
         <generate href-pipeline="add-them-extended.xpl" a="7" b="3"/>
      </li>
   </ul>
</body>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:viewport match="generate">
    <p:run>
      <p:with-input href="{/*/@href-pipeline}"/>
      <p:run-input port="source"/>
      <p:run-option name="language" select="'nl'"/>
      <p:output port="result"/>
    </p:run>
  </p:viewport>

</p:declare-step>
```

Result document:

```
<body>
   <p>Laten we optellen:</p>
   <ul>
      <li>
         <p>Als we 1 optellen bij 1 krijgen we 2!</p>
      </li>
      <li>
         <p>Als we 7 optellen bij 3 krijgen we 10!</p>
      </li>
   </ul>
</body>
```

Additional details

- What happens with any document-properties depends entirely on how the dynamically executed pipeline handles these.

Errors raised

Error code	Description
`XC0200` (pg. 219)	It is a dynamic error if the pipeline input to the p:run step is not a valid pipeline.
`XC0206` (pg. 220)	It is a dynamic error if the dynamically executed pipeline implicitly or explicitly declares a primary input port with a different name than implicitly or explicitly specified in the p:run invocation.
`XC0207` (pg. 220)	It is a dynamic error if the dynamically executed pipeline implicitly or explicitly declares a primary output port with a different name than implicitly or explicitly specified in the p:run invocation.

2.43 p:send-mail

Sends an email message.

Summary

```
<p:declare-step type="p:send-mail">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="auth" as="map(xs:string, item()+)?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:send-mail` step sends an email message. This message, described in the XML format for mail (https://datatracker.ietf.org/doc/html/draft-klyne-message-rfc822-xml-03), must appear on the `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The email message to send. The *first* document on this port must be an XML document describing the email message. It must conform to the XML format for mail (https://datatracker.ietf.org/doc/html/draft-klyne-message-rfc822-xml-03). The `<content>` element of this email message specification may contain either text or HTML. To send some other type as message body, you must leave the `<content>` element out of the first document and supply the body as a second document. Any additional documents are treated as email attachments.
`result`	`output`	`true`	`application/xml`	`false`	If the email was sent successfully, this port will emit a short XML document containing just `<c:data>true</c:data>` (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). If something went wrong during the sending operation, error `XC0162` (pg. 124) is raised.

Options:

Name	Type	Req?	Default	Description
`auth`	`map(xs:string, item()+)?`	`false`	`()`	A map containing authentication parameters. See "Authentication parameters" on page 123 for more information.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	a map containing additional information for constructing and sending the email. See "Step parameters" on page 122 for more information.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	This option controls the serialization of any documents involved in constructing the mail. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence).

Description

The `p:send-mail` step attempts to send an email message using SMTP (https://nl.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol). The address of the SMTP server and other details can be supplied using the "Step parameters" on page 122 and "Authentication parameters" on page 123.

The email message itself must be described using the XML format for mail (https://datatracker.ietf.org/doc/html/draft-klyne-message-rfc822-xml-03). Examples can be found in the specification or in the Basic usage (pg. 123) example below.

Step parameters

The `parameters` option contains additional information used for constructing and sending the email message. It is a map that associates parameters (the keys in the map) with values. The following parameters are defined:

Parameter	Datatype	Description
`send-authorization`	`xs:boolean`	If `true` (default), the authentication information provided in the "Authentication parameters" on page 123 will be used. If `false`, no authentication will be used when sending the mail.
`host`	`xs:string`	The URL of the SMTP server to use. If no value for `host` is specified, it is implementation-defined which server is used.

Parameter	Datatype	Description
`port`	`xs:unsignedShort`	The IP port to use when sending the mail to the SMTP server. If no value for `port` is specified, it is implementation-defined which port is used.

Depending on the XProc processor used, the `parameters` map may contain other, implementation-defined, keys.

Authentication parameters

The `auth` option contains additional information used for the authentication with the SMTP server. It is a map that associates parameters (the keys in the map) with values. The following authentication parameters are defined:

Parameter	Datatype	Description
`username`	`xs:string`	The username used for authentication.
`password`	`xs:string`	The password used for authentication.

Depending on the XProc processor used, the `auth` map may contain other, implementation-defined, keys.

Examples

Basic usage

Below a very basic example of how to construct an email message and use `p:send-mail` to send it. I hope I get a lot of messages like this ;).

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source">
    <p:inline>
      <emx:Message xmlns:emx="URN:ietf:params:email-xml:" xmlns:rfc822="URN:ietf:params:rfc822:">
        <rfc822:from>
          <emx:Address>
            <emx:adrs>mailto:someone@…</emx:adrs>
            <emx:name>John Doe</emx:name>
          </emx:Address>
        </rfc822:from>
        <rfc822:to>
          <emx:Address>
            <emx:adrs>mailto:erik@xatapult.nl</emx:adrs>
            <emx:name>Erik Siegel</emx:name>
          </emx:Address>
        </rfc822:to>
        <rfc822:subject>XProcRef is awesome</rfc822:subject>
        <emx:content type="text/plain" xml:space="preserve">
          Hi Erik,

          XProcRef is awesome! Congrats and kudos!

          Regards,
          John Doe
        </emx:content>
      </emx:Message>
    </p:inline>
  </p:input>

  <p:output port="result"/>

  <p:send-mail>
    <p:with-option name="auth" select="map{
      'username': '…',
      'password': '…'
    }"/>
    <p:with-option name="parameters" select="map{
      'send-authorization': true(),
      'host': '…',
      'port': …
    }"/>
  </p:send-mail>

</p:declare-step>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0159` (pg. 219)	It is a dynamic error if any key in the "`auth`" map is associated with a value that is not an instance of the required type.
`XC0160` (pg. 219)	It is a dynamic error if any key in the "`parameters`" map is associated with a value that is not an instance of the required type.
`XC0161` (pg. 219)	It is a dynamic error if the first document on the `source` port does not conform to the required format.
`XC0162` (pg. 219)	It is a dynamic error if the email cannot be send.

2.44 p:set-attributes

Add (or replace) attributes on a set of elements.

Summary

```
<p:declare-step type="p:set-attributes">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <option name="attributes" as="map(xs:QName, xs:anyAtomicType)" required="true"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
</p:declare-step>
```

The `p:set-attributes` step adds (or replaces) attributes as specified in the `attributes` option map. This is done for the element(s) matched by the `match` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to add (or replace) the attributes on.
`result`	`output`	`true`	`xml` `html`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`attributes`	`map(xs:QName, xs:anyAtomicType)`	`true`		A map with entries (attribute name, attribute value). If this map is empty, nothing will happen and the step acts as a `p:identity` (pg. 78) step.
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern that selects the element(s) to add (or replace) the attributes on. If not specified, the root element is used. This must be an XSLT match pattern that matches an element. If it matches any other kind of node, error `XC0023` (pg. 126) is raised.

Description

The `p:set-attribute` step:

- Takes the document appearing on its `source` port.
- Processes the elements that match the pattern in the `match` option:
  - If a selected element does *not* contain an attribute with a name that is a key in the `attributes` option map, an attribute with this name and a value as associated in the map is added to it.
  - If a selected element already has such an attribute, its value is replaced with the value as associated in the map.
- The resulting document appears on its `result` port.

If you just want to set a single attribute, you can also use the `p:add-attribute` (pg. 6) step.

Examples

Adding/replacing multiple attributes

This example adds the `type="special"` and a `level="2"` attributes to all `<text>` elements. One of the input `<text>` elements already has a `type` attribute, but with a different value. This existing attribute is replaced.

Source document:

```
<texts>
   <text>Hello there!</text>
   <text>This is funny…</text>
   <text type="normal">And that's normal.</text>
</texts>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:set-attributes match="text" attributes="map{'type': 'special', 'level': 2}"/>

</p:declare-step>
```

Result document:

```
<texts>
   <text type="special" level="2">Hello there!</text>
   <text type="special" level="2">This is funny…</text>
   <text type="special" level="2">And that's normal.</text>
</texts>
```

Additional details

- `p:set-attributes` preserves all document-properties of the document(s) appearing on its `source` port.
- If an attribute called `xml:base` is added or changed, the base URI of the element is updated accordingly. See also category Base URI related (pg. 212).
- You cannot use this step to add or change a namespace declaration. Attempting to do so will result in error `XC0059` (pg. 126).

  Note, however, that it is possible to add an attribute whose namespace is not in scope on the element it is added to. The XProc namespace fixup mechanism will take care of handling this and add the appropriate namespace declarations.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if that pattern matches anything other than element nodes.
`XC0059` (pg. 216)	It is a dynamic error if the QName value in the `attribute-name` option uses the prefix "`xmlns`" or any other prefix that resolves to the namespace name `http://www.w3.org/2000/xmlns/`.

2.45 p:set-properties

Sets or changes document-properties.

Summary

```
<p:declare-step type="p:set-properties">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="properties" as="map(xs:QName, item()*)" required="true"/>
  <option name="merge" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:set-properties` step sets or changes document-properties of the source document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The document to adjust the document-properties of.
`result`	`output`	`true`	`any`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`properties`	`map(xs:QName, item()*)`	`true`		A map containing the document-properties to adjust. The keys are the document-property names, the values their values.
`merge`	`xs:boolean`	`false`	`true`	If `true` (default), the supplied properties in the `properties` option are merged with the existing document-properties, replacing/overwriting existing ones. If `false`, the existing document-properties are replaced.

Description

A document flowing through an XProc pipeline carries a set of *document-properties* with it. Document-properties are key/value pairs, where the key is a QName (which, in most cases, you can treat as just a string). Their values can be anything. Getting the value(s) of document-properties can be done using the XProc functions `p:document-properties()` (https://spec.xproc.org/3.1/xproc/#f.document-properties) and `p:document-property()` (https://spec.xproc.org/3.1/xproc/#f.document-property).

XProc reserves three document-property names for its own usage: `content-type`, `base-uri` and `serialization` (see here (https://spec.xproc.org/3.1/xproc/#document-properties) for more information). However, you can also add your own and use them in any way you like.

The `p:set-properties` step can be used to change (most) document-properties. For an example of using `p:set-properties` see Converting XML to text (pg. 27) in step `p:cast-content-type` (pg. 22).

Additional details

- The `p:set-properties` step can *not* change a document media type by altering the `content-type` document-property. Any attempt to do this will result in error `XC0069` (pg. 127). To change a document media type use `p:cast-content-type` (pg. 22).
- The value of the `serialization` document-property must be a map of type `map(xs:QName, item()*)`. If not error `XD0070` (pg. 127) is raised.
- Setting a document-property called `base-uri` changes the document's base URI accordingly. See also category Base URI related (pg. 212).

Errors raised

Error code	Description
`XC0069` (pg. 217)	It is a dynamic error if the `properties` map contains a key equal to the string "`content-type`".
`XD0064` (pg. 220)	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .
`XD0070` (pg. 220)	It is a dynamic error if a value is assigned to the `serialization` document property that cannot be converted into `map(xs:QName, item()*)` according to the rules in section "QName handling" of XProc 3.0 (https://xproc.org/) .

2.46 p:sink

Discards all source documents.

Summary

```
<p:declare-step type="p:sink">
  <input port="source" primary="true" content-types="any" sequence="true"/>
</p:declare-step>
```

The `p:sink` step discards all documents that appear on its `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The document(s) to discard.

Description

The `p:sink` step eats all documents that appear on its `source` port and makes them disappear.

This step is a bit of a left-over from bygone days. In XProc 1.0, primary output ports had to be connected to something. So if you didn't need the output of some step you had to discard its results by using `p:sink`. Starting XProc 3.0 this is no longer the case: primary output ports that are not connected discard their outputs automatically. The only reason to use `p:sink` nowadays is to make this more explicit.

2.47 p:sleep

Delays the execution of the pipeline.

Summary

```
<p:declare-step type="p:sleep">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <option name="duration" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:sleep` step delays the execution of the pipeline for a specified time. Source documents are passed unchanged (like in a `p:identity` (pg. 78) step).

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The source document(s)
`result`	`output`	`true`	`any`	`true`	The resulting document(s). These will be exactly the same as what appeared on the `source` port.

Options:

Name	Type	Req?	Description
`duration`	`xs:string`	`true`	The duration of the delay, expressed as either: • A number (an `xs:double`), indicating a number of seconds. For instance `1` (1 second), `2.5` (2.5 seconds) or `0.250` (250 milliseconds). • A string that can be interpreted as an `xs:dayTimeDuration`. For instance `PT4H5M` (4 hours and 5 minutes) or `P1D` (1 day). The duration must not be negative, otherwise error `XD0036` (pg. 128) is raised.

Description

Sometimes it is useful to stop executing a pipeline for a little while, for instance when interacting with remote web servers. The `p:sleep` step does exactly this: it just stops executing the pipeline for the time as specified in the `duration` option.

Additional details

- Some XProc processors will execute steps in parallel when the flow of documents in the pipeline makes this possible (multi-threaded implementations). The `p:sleep` step is guaranteed to delay the execution of the steps that depend on its output only. Whatever happens to other steps (steps that run in parallel and do not depend on the output of the `p:sleep` invocation) is implementation-defined and therefore depends on the XProc processor used.
- A reasonable effort will be made to delay for the specified duration. However, this may not be entirely accurate.
- `p:sleep` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XD0036` (pg. 220)	It is a dynamic error if the supplied value of a variable or option cannot be converted to the required type.

2.48 p:split-sequence

Splits a sequence of documents.

Summary

```
<p:declare-step type="p:split-sequence">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="matched" primary="true" content-types="any" sequence="true"/>
  <output port="not-matched" primary="false" content-types="any" sequence="true"/>
  <option name="test" as="xs:string" required="true"/>
  <option name="initial-only" as="xs:boolean" required="false" select="false()"/>
</p:declare-step>
```

The `p:split-sequence` step takes a sequence of documents on its `source` port and divides this into two sequences, based on the evaluation expression in the `test` option and the value of the `initial-only` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The incoming sequence of documents to split.
`matched`	`output`	`true`	`any`	`true`	The documents from the `source` port for which the `test` option evaluates to `true`. If the `initial-only` option is `true`, special processing applies. See the description below.
`not-matched`	`output`	`false`	`any`	`true`	The documents from the `source` port for which the `test` option evaluates to `false`. If the `initial-only` option is `true`, special processing applies. See the description below.

Options:

Name	Type	Req?	Default	Description
`test`	`xs:string` (XPath expression)	`true`		The XPath boolean expression, as a string, that defines matching. It is evaluated with the document to test as context item (accessible with the dot operator `.`). During this evaluation, the `position()` and `last()` functions are available to get the position of the document in the sequence and the size of the sequence. See also the Using the position() and last() function (pg. 131) example.
`initial-only`	`xs:boolean`	`false`	`false`	If `false` (default), all documents for which the expression in the `test` option is `true` are considered matched and appear on the `matches` port. All documents for which this is `false` are considered not matched and appear on the `not-matched` port. If `true`, special processing applies. See the description below.

Description

The `p:split-sequence` step works like a switch in a train marshalling yard. A train with document wagons approaches the switch. The switchman has instructions to send a wagon in one direction or the other, depending on the contents of the document. In more technical terms:

- The `p:split-sequence` step takes a sequence of documents on its `source` port.
- For every document, the XPath boolean expression in the `test` option is evaluated. During this evaluation, the document is the context item (accessible with the dot operator `.`).

  The `position()` and `last()` functions are available to get the position of the document in the sequence and the size of the sequence (see also the Using the position() and last() function (pg. 131) example).
- If the `initial-only` option is `false`:
  - If the result of the `test` option evaluation is `true`, the document appears on the `matched` port.
  - If the result of the `test` option evaluation is `false`, the document appears on the `not-matched` port.
- If the `initial-only` option is `true`:
  - If the result of the `test` option evaluation is `true`, the document appears on the `matched` port, *until* a document appears for which this expression evaluates to `false`.
  - The first document for which the `test` option evaluation is `false` *and all subsequent documents* are sent to the `not-matched` port.

  In other words: it only writes the initial series of matched documents (which may be empty) to the `matched` port. All other documents are written to the `not-matched` port, irrespective of the outcome of the `test` option evaluation.

Examples

Basic usage

This example defines a sequence of `<fruit>` documents for the `source` port and splits them based on their `color` attribute. The final `p:wrap-sequence` (pg. 187) step is only there to be able to show the result as a single document.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="lemon" color="yellow"/>
    <fruit name="cauliflower" color="white"/>
  </p:input>
  <p:output port="result"/>

  <p:split-sequence test="/*/@color eq 'yellow'"/>

  <p:wrap-sequence wrapper="matched-documents"/>

</p:declare-step>
```

Result document:

```
<matched-documents>
   <fruit color="yellow" name="banana"/>
   <fruit color="yellow" name="lemon"/>
</matched-documents>
```

All the other source documents appear on the `not-matched` port, as the following example proofs:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="lemon" color="yellow"/>
    <fruit name="cauliflower" color="white"/>
  </p:input>
  <p:output port="result"/>

  <p:split-sequence test="/*/@color eq 'yellow'"/>

  <p:wrap-sequence wrapper="not-matched-documents">
    <p:with-input pipe="not-matched"/>
  </p:wrap-sequence>

</p:declare-step>
```

Result document:

```
<not-matched-documents>
   <fruit color="orange" name="orange"/>
   <fruit color="white" name="cauliflower"/>
</not-matched-documents>
```

Using the initial-only option

This example shows what happens to the Basic usage (pg. 130) example when we set the `initial-only` option to `true`. The second source document (about oranges) does *not* match, so this *and* all subsequent documents are considered not-matched and sent to the `not-matched` port. Only the first source document (about bananas) appears on the `matched` port.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="lemon" color="yellow"/>
    <fruit name="cauliflower" color="white"/>
  </p:input>
  <p:output port="result"/>

  <p:split-sequence test="/*/@color eq 'yellow'" initial-only="true"/>

  <p:wrap-sequence wrapper="matched-documents"/>

</p:declare-step>
```

Result document:

```
<matched-documents>
   <fruit color="yellow" name="banana"/>
</matched-documents>
```

Using the position() and last() function

The following example shows that, during evaluation of the `test` option, the `position()` and `last()` functions are available to get the position of the document in the sequence and the size of the sequence. It uses this to match the last document only.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="lemon" color="yellow"/>
    <fruit name="cauliflower" color="white"/>
  </p:input>
  <p:output port="result"/>

  <p:split-sequence test="position() eq last()"/>

  <p:wrap-sequence wrapper="matched-documents"/>

</p:declare-step>
```

Result document:

```
<matched-documents>
   <fruit color="white" name="cauliflower"/>
</matched-documents>
```

Additional details

- `p:split-sequence` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0150` (pg. 219)	It is a dynamic error if evaluating the XPath expression in option `test` on a context document results in an error.

2.49 p:store

Stores a document.

Summary

```
<p:declare-step type="p:store">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <output port="result-uri" primary="false" content-types="application/xml" sequence="false"/>
  <option name="href" as="xs:anyURI" required="true"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:store` step stores the document appearing on its `source` port to a URI. This document is passed unchanged to the `result` port. It outputs the absolute URI of the location of the stored document on the `result-uri` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The document to store.
`result`	`output`	`true`	`any`	`false`	The resulting document. This will be exactly the same as the document on the `source` port.
`result-uri`	`output`	`false`	`application/xml`	`false`	An XML document consisting of just a single `<c:result>` element containing the *absolute* URI the document is stored to (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). Example: `<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:///some/path/document.xml</c:result>`

Options:

Name	Type	Req?	Default	Description
`href`	`xs:anyURI`	`true`		The URI to store the document to. In most cases, `p:store` will be used to store a document to disk. An absolute URI for this must start with `file://`. For instance, on Windows, `file:///C:/some/path/document.xml` (although Windows uses backslashes (\) to separate path components, slashes (/) work fine and are more universal). Using a single slash after `file:` also works: `file:/C:/some/path/document.xml`. An attempt will be made to create all non-existing folders in the path. If this value is relative, it is resolved against the base URI of the element on which this option is specified. In most cases this will be the static base URI of your pipeline (the path where the XProc source containing the `p:store` step is stored).
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/) for storing the document. If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence). Example: `serialization="map{'indent': true()}"`

Description

The `p:store` step stores the document appearing on its `source` port to a URI. This document is passed unchanged to the `result` port. So within the pipeline the step acts as a `p:identity` (pg. 78) step.

It outputs the absolute URI of the location of the stored document on the `result-uri` port.

Examples

Basic usage

Whatever input document is passed to the following pipeline, it is stored to disk in a document `tmp/x.xml`, relative to where the pipeline is stored. The final `p:identity` (pg. 78) step is just used to show you the document appearing on the `result-uri` port.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:store href="tmp/x.xml"/>

  <p:identity>
    <p:with-input pipe="result-uri"/>
  </p:identity>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">file:/…/…/x.xml</c:result>
```

Doing something with what appears on the `result-uri` port is of course completely optional. If you don't attach anything to this port, the `<c:result>` document will simply disappear into oblivion.

Using p:store for writing intermediate results

When developing a pipeline, you often want to take a look at what exactly is flowing through it at a certain stage. Inserting a (temporary) `p:store` step is the most easy way to quickly write some intermediate result to a temporary file on disk for inspection. Since `p:store` acts like a `p:identity` (pg. 78) step. nothing happens to the flow in your pipeline.

```
…
<p:store href="file:///my/debug/files/location/stepx.xml"/>
…
```

Sometimes you want to keep these temporary storage steps after development. You never know what bugs will pop up and you might need them again! XProc has static options that you can use for this:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">
  <p:option static="true" name="write-debug-documents" select="true()"/> … <p:store use-when="{$write-debug-
documents}"
    href="file:///my/debug/files/location/stepx.xml"/> … </p:declare-step>
```

For the production version you set the `write-debug-documents` static option to `false()`. This makes the `p:store` disappear from the processed code, as if it was never there…

Additional details

- `p:store` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0050` (pg. 216)	It is a dynamic error the file or directory cannot be copied to the specified location.

2.50 p:string-replace

Replaces nodes with strings.

Summary

```
<p:declare-step type="p:string-replace">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <option name="match" as="xs:string" required="true"/>
  <option name="replace" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:string-replace` step takes the document appearing on its `source` port and replaces nodes matching the `match` option with a string that is computed using the XPath expression in the `replace` option.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to replace the nodes in.

`result`	`output`	`true`	`text`
			`xml`
			`html`
				`false`	The resulting document.

Options:

Name	Type	Req?	Description
`match`	`xs:string` (XSLT selection pattern)	`true`	The XSLT match pattern for the nodes to replace, as a string.
`replace`	`xs:string` (XPath expression)	`true`	An XPath expression whose result will replace the nodes matched by the `match` option. During the evaluation of this expression, the context item is the matched node (accessible with the dot operator `.`).

Description

The `p:string-replace` step does the following:

- It takes the document appearing on its `source` port and holds the XSLT selection pattern in the `match` option against this.
- For each node matched:
  - The matched node becomes the context item (accessible with the dot operator `.`).
  - The XPath expression in the `replace` option is evaluated and the result is turned into a string.
  - If the matched node is an attribute, the *value* of the attribute is replaced with this string.
  - If the document-node is matched, the full document will be replaced by the string (and the result will therefore be a text document).
  - In all other cases, the full node is replaced by the string.

So this step replaces the matched node(s) with the result of a *dynamically evaluated* expression. This doesn't mean this expression can't be a constant: see the Basic usage (pg. 135) example. However, it allows you to do all kinds of nifty calculations, based on where the match is: see the Advanced usage (pg. 136) example.

This step replaces matched nodes with strings. If you need to replace matched nodes with full documents, have a look at the `p:replace` (pg. 115) step.

Examples

Basic usage

The following example simply replaces the thing's `<contents>` element with a (constant) description:

Source document:

```
<thing>
   <contents/>
</thing>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:string-replace match="thing/contents" replace="'This is a thing of beauty!'"/>

</p:declare-step>
```

Result document:

```
<thing>
      This is a thing of beauty!
   </thing>
```

Please notice the single quotes around the value in the `replace` option. This option must not hold just some value but an XPath *expression*. This means that if you need a constant string, you need to write it as an XPath string, therefore the single quotes.

If the `match` option matches an attribute, the *value* of the attribute is replaced:

Source document:

```
<thing description="notfilledinyet"/>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:string-replace match="thing/@description" replace="'This is a thing of beauty!'"/>

</p:declare-step>
```

Result document:

```
<thing description="This is a thing of beauty!"/>
```

Advanced usage

The following example fills empty `description` attributes with a value based on the index/occurrence of the parent `<thing>` element and the value of its `name` attribute:

Source document:

```
<things>
   <thing name="brick" description=""/>
   <thing name="mortar" description=""/>
   <thing name="door" description="A door"/>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:string-replace match="thing/@description[. eq '']" replace="'Thing ' || count(../preceding-
sibling::thing) + 1 || ': ' || ../@name"/>

</p:declare-step>
```

Result document:

```
<things>
   <thing name="brick" description="Thing 1: brick"/>
   <thing name="mortar" description="Thing 2: mortar"/>
   <thing name="door" description="A door"/>
</things>
```

Notice that to be able to perform this trick, the `description` attribute must already be there! A `<thing>` element without such an attribute will not be changed. So if you have content where this is lacking, you'll need to prepare it. In this case we use the `p:add-attribute` (pg. 6) step to add a `description` attribute to any `<thing>` element that is lacking one first:

Source document:

```
<things>
   <thing name="brick" description=""/>
   <thing name="screw"/>
   <thing name="mortar" description=""/>
   <thing name="door" description="A door"/>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:add-attribute match="thing[empty(@description)]" attribute-name="description" attribute-value=""/>
  <p:string-replace match="thing/@description[. eq '']" replace="'Thing ' || count(../preceding-
sibling::thing) + 1 || ': ' || ../@name"/>

</p:declare-step>
```

Result document:

```
<things>
   <thing name="brick" description="Thing 1: brick"/>
   <thing description="Thing 2: screw" name="screw"/>
   <thing name="mortar" description="Thing 3: mortar"/>
   <thing name="door" description="A door"/>
</things>
```

Using p:with-option for the replace option

Options can also be set using the `<p:with-option>` element. If you use this for the `replace` option, make sure that you pass the expression in the `replace` option *as a string*. If you don't, it will get evaluated by the pipeline, before the invocation of the `p:string-replace` step, and that is very probably not what you intend. Here is an example of how to do this right, based on the first example of Advanced usage (pg. 136):

Source document:

```
<things>
   <thing name="brick" description=""/>
   <thing name="mortar" description=""/>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:string-replace match="thing/@description">
    <p:with-option name="replace" select="'''Thing '' || count(../preceding-
sibling::thing) + 1 || '': '' || ../@name'"/>
  </p:string-replace>

</p:declare-step>
```

Result document:

```
<things>
   <thing name="brick" description="Thing 1: brick"/>
   <thing name="mortar" description="Thing 2: mortar"/>
</things>
```

If you accidentally write the `p:with-option/@select` as the value of the `p:string-replace/@replace` attribute in Advanced usage (pg. 136) (an easy and probable mistake to make), the XProc processor will raise an error:

- The expression in the `p:with-option/@select` is executed by the pipeline first and results in: `'Thing 1:'`
- This is *not* a valid XPath expression…
- The `p:string-replace` step tries to evaluate `Thing 1:` as an expression and fails miserably (but rightly).

Additional details

- `p:string-replace` preserves all document-properties of the document(s) appearing on its `source` port. There is one exception: if the resulting document contains only text, the `content-type` document-property is changed to `text/plain` and the `serialization` document-property is removed.
- If an attribute called `xml:base` is added or changed, the base URI of the element is updated accordingly. See also category Base URI related (pg. 212).

2.51 p:text-count

Counts the number of lines in a text document.

Summary

```
<p:declare-step type="p:text-count">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
</p:declare-step>
```

The `p:text-count` step counts the number of lines in the text document appearing on its `source` port and returns an XML document on its `result` port containing that number.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The text document to count the lines of.
`result`	`output`	`true`	`application/xml`	`false`	An XML document consisting of a single `<c:result>` element containing the number of lines (the `c` prefix here is bound to the `http://www.w3.org/ns/xproc-step` namespace). Example: `<c:result xmlns:c="http://www.w3.org/ns/xproc-step">6</c:result>`

Description

The `p:text-count` step simply counts the number of lines in the text document appearing on its `source` port. This number is returned on the `result` port, wrapped in an `<c:result>` element.

Examples

Basic usage

Assume we have a text document, called `lines.txt`, that looks like this and we want to count the number of lines using `p:text-count`:

```
line 1
line 2
line 3
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-count/>

</p:declare-step>
```

Result document:

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">3</c:result>
```

Additional details

- No document-properties from the document on the `source` port survives. The resulting document has a `content-type` document-property set to `application/xml` and no `base-uri` document-property.
- What exactly constitutes a line-end is defined in the XML specification (https://www.w3.org/TR/xml/#sec-line-ends).
- If the very last character of the source document is a line-end, this is ignored. So it does not count as a separator between that line and a following line (that contains no characters).

2.52 p:text-head

Returns lines from the beginning of a text document.

Summary

```
<p:declare-step type="p:text-head">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="text" sequence="false"/>
  <option name="count" as="xs:integer" required="true"/>
</p:declare-step>
```

The `p:text-head` step returns lines from the beginning of a text document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The text document to get the lines from.
`result`	`output`	`true`	`text`	`false`	The resulting lines.

Options:

Name	Type	Req?	Description
`count`	`xs:integer`	`true`	Indicates what `p:text-head` should do: • If positive, `p:text-head` returns the first `count` lines. • If zero, `p:text-head` returns all lines. • If negative, `p:text-head` returns all lines *except* the first `count` lines.

Description

The `p:text-head` step takes lines from the beginning of the text document appearing on its `source` port and returns these lines as a text document on its `result` port. What exactly happens depends on the value of the `count` option.

As you might have guessed there is also a step that returns lines from the *end* of a document: `p:text-tail` (pg. 146).

Examples

Basic usage

Assume we have a text document, called `lines.txt`, that looks like this and we want to get the first 2 lines using `p:text-head`:

```
line 1
line 2
line 3
line 4
line 5
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-head count="2"/>

</p:declare-step>
```

Result document:

```
line 1
line 2
```

Setting the `count` option to `0` will simply return the original document (the step now acts like a `p:identity` (pg. 78) step).

Setting the `count` option to `-2` will return all lines *except* the first two:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-head count="-2"/>

</p:declare-step>
```

Result document:

```
line 3
line 4
line 5
```

Additional details

- `p:text-head` preserves all document-properties of the document(s) appearing on its `source` port.
- What exactly constitutes a line-end is defined in the XML specification (https://www.w3.org/TR/xml/#sec-line-ends).
- All lines returned by `p:text-head` are terminated with a line-end character (line-feed, `
`).

2.53 p:text-join

Concatenates text documents.

Summary

```
<p:declare-step type="p:text-join">
  <input port="source" primary="true" content-types="text" sequence="true"/>
  <output port="result" primary="true" content-types="text" sequence="false"/>
  <option name="override-content-type" as="xs:string?" required="false" select="'text/plain'"/>
  <option name="prefix" as="xs:string?" required="false" select="()"/>
  <option name="separator" as="xs:string?" required="false" select="()"/>
  <option name="suffix" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:text-join` step takes the document(s) appearing on its `source` port and concatenates these.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`true`	The sequence of text documents to concatenate.
`result`	`output`	`true`	`text`	`false`	The resulting text document.

Options:

Name	Type	Req?	Default	Description
`override-content-type`	`xs:string?`	`false`	`text/plain`	The media type of the result document (the value for the result `content-type` document-property). This must be a text media type (`text/*`).
`prefix`	`xs:string?`	`false`	`()`	A prefix string for the result document (also used when there are no documents on the `source` port).
`separator`	`xs:string?`	`false`	`()`	A separator string to insert in between adjacent documents.
`suffix`	`xs:string?`	`false`	`()`	A suffix string for the result document (also used when there are no documents on the `source` port).

Description

The `p:text-join` step takes the document(s) appearing on its `source` port and concatenates these (in order of appearance).

Using the `separator`, `prefix` and `suffix` options it is possible to insert additional strings in between the documents, before the first document, or after the last document.

Examples

Basic usage

Assume we have three source text documents:

- `to-join-01.txt`

```
First document to join!
```

- `to-join-02.txt`

```
Second document to join! It's getting better…
```

- `to-join-03.txt`

```
Third document to join! Last but not least!
```

Straight concatenation using `p:text-join` looks like this:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="to-join-1.txt"/>
    <p:document href="to-join-2.txt"/>
    <p:document href="to-join-3.txt"/>
  </p:input>
  <p:output port="result"/>

  <p:text-join/>

</p:declare-step>
```

Result document:

```
First document to join!
Second document to join! It's getting better…
Third document to join! Last but not least!
```

And this is what happens if we use the `separator`, `prefix` and `suffix` options:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:document href="to-join-1.txt"/>
    <p:document href="to-join-2.txt"/>
    <p:document href="to-join-3.txt"/>
  </p:input>
  <p:output port="result"/>

  <p:text-join separator="=========&#xA;" prefix="==START==&#xA;" suffix="==END==&#xA;"/>

</p:declare-step>
```

Result document:

```
==START==
First document to join!
=========
Second document to join! It's getting better…
=========
Third document to join! Last but not least!
==END==
```

Notice the use of the line-end character (line-feed, `
`) in the option values. This will cause the inserted strings to become separate lines.

When there are no documents on the `source` port, the `prefix` and `suffix` options still apply:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <p:empty/>
  </p:input>
  <p:output port="result"/>

  <p:text-join separator="=========&#xA;" prefix="==START==&#xA;" suffix="==END==&#xA;"/>

</p:declare-step>
```

Result document:

```
==START==
==END==
```

Additional details

- No document-properties from the source document(s) survive. The joined document has no `base-uri` document-property.
- This operation does not require identifying lines. Therefore, no special end-of-line handling is performed.

Errors raised

Error code	Description
`XC0001` (pg. 216)	It is a dynamic error if the value of option `override-content-type` is not a text media type.
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.54 p:text-replace

Replace substrings in a text document.

Summary

```
<p:declare-step type="p:text-replace">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="text" sequence="false"/>
  <option name="pattern" as="xs:string" required="true"/>
  <option name="replacement" as="xs:string" required="true"/>
  <option name="flags" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:text-replace` step takes the text document appearing on its `source` port and replaces substrings that match a regular expression with a replacement string.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The text document to replace the substrings in.
`result`	`output`	`true`	`text`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`pattern`	`xs:string`	`true`		The XPath regular expression that matches the substrings to replace. See the `$pattern` argument of the XPath function replace() (https://w3.org/TR/xpath-functions-31/#func-replace) for more details.
`replacement`	`xs:string`	`true`		The replacement string. See the `$replacement` argument of the XPath function replace() (https://w3.org/TR/xpath-functions-31/#func-replace) for more details.
`flags`	`xs:string?`	`false`	`()`	Flags governing the matching process. See the description of regular expression flags (https://www.w3.org/TR/xpath-functions-31/#flags) in the XPath standard for more details.

Description

The `p:text-replace` step is "just" a convenience wrapper around the Xpath replace() (https://w3.org/TR/xpath-functions-31/#func-replace) function. This function takes a string and replaces substrings that match a regular expression with a replacement string. For the `p:text-replace` step, the input string is the full document on the `source` port. The resulting document appears on the `result` port.

Examples

Basic usage

Assume we have a text document, called `lines.txt`, that looks like this and we want to replace all `t` characters followed by an `h` or an `e` with two * characters (case-insensitive):

```
This is an example of a very simple text file… But simple is often the best!
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-replace pattern="t[h|e]" replacement="**" flags="i"/>

</p:declare-step>
```

Result document:

```
**is is an example of a very simple **xt file… But simple is of**n **e best!
```

Additional details

- `p:text-replace` preserves all document-properties of the document(s) appearing on its `source` port.
- This operation does not require identifying lines. Therefore, no special end-of-line handling is performed.

Errors raised

Error code	Description
`XC0147` (pg. 219)	It is a dynamic error if the specified value is not a valid XPath regular expression.

2.55 p:text-sort

Sorts lines in a text document.

Summary

```
<p:declare-step type="p:text-sort">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="text" sequence="false"/>
  <option name="case-order" as="xs:string?" required="false" select="()" values="('upper-first', 'lower-
first')"/>
  <option name="collation" as="xs:string" required="false" select="'https://www.w3.org/2005/xpath-functions/
collation/codepoint'"/>
  <option name="lang" as="xs:language?" required="false" select="()"/>
  <option name="order" as="xs:string" required="false" select="'ascending'" values="('ascending', 'descending')"/
>
  <option name="sort-key" as="xs:string" required="false" select="'.'"/>
  <option name="stable" as="xs:boolean" required="false" select="true()"/>
</p:declare-step>
```

The `p:text-sort` step sorts lines in the text document appearing on its `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The text document to sort the lines of.
`result`	`output`	`true`	`text`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`case-order`	`xs:string?`	`false`	`()`	Defines whether upper-case characters are considered to come before or after lower-case characters. Must have a value `upper-first` or `lower-first`. If not provided, its value is language-dependent. This option is only used if no value is available for the `collation` option.
`collation`	`xs:string`	`false`	`https://www.w3.org/2005/xpath-functions/collation/codepoint`	The collation to use for sorting. The only collation that is always supported is the `Unicode codepoint collation (https://www.w3.org/2005/xpath-functions/collation/codepoint/)`. This is also the default value for this option. Whether any other collations are supported is implementation-defined and therefore dependent on the XProc processor used.
`lang`	`xs:language?`	`false`	`()`	The language to sort the lines for. This influences, for instance, the order of accented characters. Its default value is implementation-defined and therefore dependent on the XProc processor used. A value for the `lang` option must be a valid language code according to RFC 4646 (tags for identifying languages) (https://www.ietf.org/rfc/rfc4646.txt). For instance: `en-us` or `nl-nl`. This option is only used if no value is available for the `collation` option.
`order`	`xs:string`	`false`	`ascending`	The sort order, either `ascending` (default) or `descending`.
`sort-key`	`xs:string` (XPath expression)	`false`	`.`	An XPath expression that results in the string to sort the lines on. It is evaluated for each line, with the line to sort (as a string) as context item (accessible with the dot operator `.`). During this evaluation, the `position()` and `last()` functions are available to get the position of the line in the document and the number of lines. See Reversing the line order (pg. 145) for an example of using these functions here.
`stable`	`xs:boolean`	`false`	`true`	This option tells the sorting algorithm what to do with lines with same sorting key. If its value is `true` (default), these lines are retained in their original order. If `false`, there is no need to this and the algorithm *may* change their order (but not necessarily so).

Description

The `p:text-sort` step takes the text document appearing on its `source` port and turns this into lines. These lines are then sorted according to the values of the step options. This sort process is the same as described for the XSLT `xsl:sort (https://www.w3.org/TR/xslt-30/#xsl-sort)` instruction. The result appears on the `result` port.

Examples

Basic usage

Assume we have a text document, called `lines.txt`, that looks like this and we want to sort the lines using `p:text-sort`:

```
XProc steps rock!
An important addition to our XML processing toolkit.
What a joy!
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-sort/>

</p:declare-step>
```

Result document:

```
An important addition to our XML processing toolkit.
What a joy!
XProc steps rock!
```

Reversing the line order

This example is not very useful in itself, but it shows the use of the `position()` and `last()` function in the `sort-key` option. We set this option to `last() - position()`, which has the effect of *reversing* the line order.

Source document (`lines-2.txt`):

```
line 1
line 2
line 3
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines-2.txt"/>
  <p:output port="result"/>

  <p:text-sort sort-key="last() - position()"/>

</p:declare-step>
```

Result document:

```
line 3
line 2
line 1
```

Additional details

- `p:text-sort` preserves all document-properties of the document(s) appearing on its `source` port.
- What exactly constitutes a line-end is defined in the XML specification (https://www.w3.org/TR/xml/#sec-line-ends).
- All lines returned by `p:text-sort` are terminated with a line-end character (line-feed, `
`).

Errors raised

Error code	Description
`XC0098` (pg. 217)	It is a dynamic error if a dynamic XPath error occurred while applying sort-key to a line.
`XC0099` (pg. 217)	It is a dynamic error if the result of applying `sort-key` to a given line results in a sequence with more than one item.

2.56 p:text-tail

Returns lines from the end of a text document.

Summary

```
<p:declare-step type="p:text-tail">
  <input port="source" primary="true" content-types="text" sequence="false"/>
  <output port="result" primary="true" content-types="text" sequence="false"/>
  <option name="count" as="xs:integer" required="true"/>
</p:declare-step>
```

The `p:text-tail` step returns lines from the end of a text document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text`	`false`	The text document to get the lines from.
`result`	`output`	`true`	`text`	`false`	The resulting lines.

Options:

Name	Type	Req?	Description
`count`	`xs:integer`	`true`	Indicates what `p:text-tail` should do: • If positive, `p:text-tail` returns the last `count` lines. • If zero, `p:text-tail` returns all lines. • If negative, `p:text-tail` returns all lines *except* the last `count` lines.

Description

The `p:text-tail` step takes lines from the end of the text document appearing on its `source` port and returns these lines as a text document on its `result` port. What exactly happens depends on the value of the `count` option.

As you might have guessed there is also a step that returns lines from the *beginning* of a document: `p:text-head` (pg. 139).

Examples

Basic usage

Assume we have a text document, called `lines.txt`, that looks like this and we want to get the last 2 lines using `p:text-tail`:

```
line 1
line 2
line 3
line 4
line 5
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-tail count="2"/>

</p:declare-step>
```

Result document:

```
line 4
line 5
```

Setting the `count` option to `0` will simply return the original document (the step now acts like a `p:identity` (pg. 78) step).

Setting the `count` option to `-2` will return all lines *except* the last two:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="lines.txt"/>
  <p:output port="result"/>

  <p:text-tail count="-2"/>

</p:declare-step>
```

Result document:

```
line 1
line 2
line 3
```

Additional details

- `p:text-tail` preserves all document-properties of the document(s) appearing on its `source` port.
- What exactly constitutes a line-end is defined in the XML specification (https://www.w3.org/TR/xml/#sec-line-ends).
- All lines returned by `p:text-tail` are terminated with a line-end character (line-feed, `
`).

2.57 p:unarchive

Extracts documents from an archive file.

Summary

```
<p:declare-step type="p:unarchive">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <option name="exclude-filter" as="xs:string*" required="false" select="()"/>
  <option name="format" as="xs:QName?" required="false" select="()"/>
  <option name="include-filter" as="xs:string*" required="false" select="()"/>
  <option name="override-content-types" as="array(array(xs:string))?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="relative-to" as="xs:anyURI?" required="false" select="()"/>
</p:declare-step>
```

The `p:unarchive` extracts document from an archive file (for instance a ZIP file) and returns these on its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The archive to extract the documents from.
`result`	`output`	`true`	`any`	`true`	The extracted documents.

Options:

Name	Type	Req?	Default	Description
`exclude-filter`	`xs:string*` (XPath regular expression)	`false`	`()`	A sequence of XPath regular expressions (as strings) that determine which files in the archive are *not* extracted. See "Determining which files to extract" on page 149.
`format`	`xs:QName?`	`false`	`()`	The format of the archive file on the `source` port: • If its value is `zip`, the `p:unarchive` step expects a ZIP archive on the `source` port. • If absent or the empty sequence, the `p:unarchive` step tries to guess the archive file format. The only format that this step is required to recognize and handle is ZIP. • Whether any other archive formats can be handled and what their names (values for this option) are depends on the XProc processor used.
`include-filter`	`xs:string*` (XPath regular expression)	`false`	`()`	A sequence of XPath regular expressions (as strings) that determine which files in the archive are extracted. See "Determining which files to extract" on page 149.
`override-content-types`	`array(array (xs:string))?`	`false`	`()`	Use this to override the content-type determination for the extracted files (the value of their `content-type` document-property). This mechanism works the same as for the `p:archive-manifest` (pg. 18) step. See Overriding content types (pg. 151) for an example.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the document extraction. The XProc specification does not define any parameters for this option. A specific XProc processor might define its own.
`relative-to`	`xs:anyURI?`	`false`	`()`	This option can be used to explicitly set the `base-uri` document-property of the extracted documents. See "The base URI of the extracted files" on page 149

Description

The `p:unarchive` step allows you to extract one or more documents from an archive (for instance a ZIP file). The result will be a sequence of the extracted documents on the `result` port.

- You can specify exactly which documents to extract or not to extract using the `include-filter` and `exclude-filter` options. See "Determining which files to extract" on page 149.
- Sometimes it is important to specify the exact base URI of the extracted documents for subsequent steps. You can do this using the `relative-to` option. See "The base URI of the extracted files" on page 149.
- Although probably rare, it is also possible to control the content type (MIME type) of the extracted documents, using the `override-content-types` option. See Overriding content types (pg. 151) for an example.

Archives come in many formats. The only format the `p:unarchive` step is required to handle is ZIP. However, depending on the XProc processor used, other formats may also be processed.

Determining which files to extract

The `include-filter` and `exclude-filter` options determine which documents to extract. Both option must be a sequence of zero or more XPath regular expressions, as strings. For an example see Excluding documents (pg. 150). Basic operation:

- The paths of the documents *in* the archive are matched against the regular expressions.
- A document must be included and not excluded.
- An empty `include-filter` option means: *all* documents are included. An empty `exclude-filter` option means: *no* documents are excluded.

In more detail:

- First, the `include-filter` option is processed:
  - If it is empty (its value is the empty sequence `()`), *all* documents in the archive are included.
  - Otherwise, the path of every document *in* the archive is matched against the list of regular expressions in the `include-filter` option (like in `matches($path-in-archive, $regular-expression)`). If one of the regular expression matches, the document is included, otherwise it is excluded.
- Now the `exclude-filter` option is processed against the resulting list of entries:
  - If it is empty (its value is the empty sequence `()`), *no* further documents excluded.
  - Otherwise, the path of every document *in* the archive is matched against the list of regular expressions in the `exclude-filter` option (like in `matches($path-in-archive, $regular-expression)`). If one of the regular expression matches, the document is excluded, otherwise it is included.

If the value for one of these options is a sequence with just a single values, you can set this by attribute:

```
<p:unarchive exlude-filter="\.xml$"/>
```

However, if more than one value is involved you *must* use `<p:with-option>` (providing a sequence with multiple values by attribute is not possible):

```
<p:unarchive>
  <p:with-option name="exclude-filter" select="('\.xml$', '\.jpg$')"/>
</p:unarchive>
```

The base URI of the extracted files

The `relative-to` option can be used to specify the `base-uri` document-property of the extracted documents:

- If the `relative-to` option is *not* specified, the `base-uri` document-property of an extracted document is the full URI of the archive followed by the path of the document *in* the archive. For instance: `file:///path/to/archive/archive.zip/path/in/archive/test.xml`.
- If a `relative-to` option is specified, it must be a valid URI. The `base-uri` document-property of an extracted document is this URI followed by the path of the document *in* the archive. For instance, assume we've set the `relative-to` option to `file:///my/documents/`: `file:///my/documents/path/in/archive/test.xml`.

The Basic usage (pg. 149) and most other examples show what happens if you don't specify a `relative-to` option. The Using relative-to (pg. 151) example shows what happens if you do.

Examples

Basic usage

Assume we have a simple ZIP archive with two entries:

- An XML file in the root called `reference.xml`
- An image in an `images/` sub-directory called `logo.png`.

The following pipeline uses `p:unarchive` to extract its contents. The `<p:for-each>` construction after the `p:unarchive` creates an overview of what was extracted. The actual extracted files are discarded.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unarchive/>

  <p:for-each>
    <p:identity>
      <p:with-input exclude-inline-prefixes="#all">
        <unarchived-file href="{p:document-property(/, 'base-uri')}" content-type="{p:document-
property(/, 'content-type')}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="unarchived-files"/>

</p:declare-step>
```

Resulting overview of the extracted files:

```
<unarchived-files>
   <unarchived-file content-type="image/png" href="file:/…/…/test.zip/images/logo.png"/>
   <unarchived-file content-type="application/xml" href="file:/…/…/test.zip/reference.xml"/>
</unarchived-files>
```

Excluding documents

This example uses the same ZIP archive as Basic usage (pg. 149). The `exclude-filter` option excludes the entries ending with `.xml`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unarchive exclude-filter="\.xml$"/>

  <p:for-each>
    <p:identity>
      <p:with-input exclude-inline-prefixes="#all">
        <unarchived-file href="{p:document-property(/, 'base-uri')}" content-type="{p:document-
property(/, 'content-type')}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="unarchived-files"/>

</p:declare-step>
```

Resulting overview of the extracted files:

```
<unarchived-files>
   <unarchived-file content-type="image/png" href="file:/…/…/test.zip/images/logo.png"/>
</unarchived-files>
```

The following example excludes all documents from the `images` sub-directory:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unarchive exclude-filter="^images/"/>

  <p:for-each>
    <p:identity>
      <p:with-input exclude-inline-prefixes="#all">
        <unarchived-file href="{p:document-property(/, 'base-uri')}" content-type="{p:document-
property(/, 'content-type')}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="unarchived-files"/>

</p:declare-step>
```

Resulting overview of the extracted files:

```
<unarchived-files>
   <unarchived-file content-type="application/xml" href="file:/…/…/test.zip/reference.xml"/>
</unarchived-files>
```

Using relative-to

This example uses the same ZIP archive as Basic usage (pg. 149). The following pipeline explicitly sets the base part of the URIs for the extracted documents to `file:///my/documents/`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unarchive relative-to="file:///my/documents/"/>

  <p:for-each>
    <p:identity>
      <p:with-input exclude-inline-prefixes="#all">
        <unarchived-file href="{p:document-property(/, 'base-uri')}" content-type="{p:document-
property(/, 'content-type')}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="unarchived-files"/>

</p:declare-step>
```

Resulting overview of the extracted files:

```
<unarchived-files>
   <unarchived-file content-type="image/png" href="file:///my/documents/images/logo.png"/>
   <unarchived-file content-type="application/xml"
                    href="file:///my/documents/reference.xml"/>
</unarchived-files>
```

Overriding content types

This example uses the same ZIP archive as Basic usage (pg. 149). The following pipeline explicitly sets the content type for `.png` files to `application/octet-stream`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unarchive>
    <p:with-option name="override-content-types" select="[ ['\.png$', 'application/octet-stream'] ]"/>
  </p:unarchive>

  <p:for-each>
    <p:identity>
      <p:with-input exclude-inline-prefixes="#all">
        <unarchived-file href="{p:document-property(/, 'base-uri')}" content-type="{p:document-
property(/, 'content-type')}"/>
      </p:with-input>
    </p:identity>
  </p:for-each>
  <p:wrap-sequence wrapper="unarchived-files"/>

</p:declare-step>
```

Resulting overview of the extracted files:

```
<unarchived-files>
   <unarchived-file content-type="application/octet-stream"
                    href="file:/…/…/test.zip/images/logo.png"/>
   <unarchived-file content-type="application/xml" href="file:/…/…/test.zip/reference.xml"/>
</unarchived-files>
```

More information about how this mechanism works can be found in the description of the `p:archive-manifest` (pg. 18) step.

Additional details

- No document-properties from the document on the `source` port survive.
- A relative value for the `relative-to` option gets de-referenced against the base URI of the element in the pipeline it is specified on. In most cases this will be the path of the pipeline document.
- Paths in an archive are always relative. However, depending on how archives are constructed, a path in an archive can be with or without a leading `/`. Usually it will be without. For archives constructed by `p:archive` (pg. 11) no leading slash will be present.
- The only format this step is required to handle is ZIP. The ZIP format definition can be found here (https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT).

Errors raised

Error code	Description
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XC0085` (pg. 217)	It is a dynamic error if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.
`XC0120` (pg. 218)	It is a dynamic error if the `relative-to` option is not present and the document on the `source` port does not have a base URI.
`XC0147` (pg. 219)	It is a dynamic error if the specified value is not a valid XPath regular expression.

2.58 p:uncompress

Uncompresses a document.

Summary

```
<p:declare-step type="p:uncompress">
  <input port="source" primary="true" content-types="any" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="content-type" as="xs:string" required="false" select="'application/octet-stream'"/>
  <option name="format" as="xs:QName?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:uncompress` step expects on its `source` port a compressed (usually binary) document. It outputs an uncompressed version of this document on its `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`false`	The document to uncompress
`result`	`output`	`true`	`any`	`false`	The resulting uncompressed document.

Options:

Name	Type	Req?	Default	Description
`content-type`	`xs:string`	`false`	`application/octet-stream`	Specifies the media type for the resulting uncompressed document that appears on the `result` port (and therefore the value of its `content-type` document-property). If not specified, this media type will become `application/octet-stream` (the generic media type for "any binary document").
`format`	`xs:QName?`	`false`	`()`	Specifies the compression format of the source document: • If its value is the empty sequence (default), `p:uncompress` tries to guess this format by inspecting the document's `content-type` document-property and/or inspecting its contents. How this is done is implementation-defined and therefore dependent on the XProc processor used. A GZIP (https://datatracker.ietf.org/doc/html/rfc1952) compressed document should be recognized. Whether any other format is recognized is also implementation-defined. • If its value is `gzip` the step expects a document compressed according to the GZIP (https://datatracker.ietf.org/doc/html/rfc1952) specification. • Support for any other value is implementation-defined and therefore dependent on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the uncompression. Keys, values and their meaning depend on the value of the `method` option and the XProc processor used.

Description

The `p:uncompress` step uncompresses the document appearing on its `source` port.

To do this it first determines the compression format (see the description of the `format` option). Usually this will be GZIP (https://datatracker.ietf.org/doc/html/rfc1952), the only compression format an XProc processor is required to support.

After the uncompression, the result is interpreted according to the value of the step's `content-type` option. If nothing is specified, the resulting document will just flow out as a generic binary document (media type `application/octet-stream`). However, if an explicit media type is specified, for instance `text/xml`, the result is interpreted as such and cast to this type. Of course this must be possible, if not error `XC0201` (pg. 154) is raised.

Additional details

- `p:uncompress` preserves all document-properties of the document(s) appearing on its `source` port. Exception is the `content-type` document-property which is updated accordingly.

Errors raised

Error code	Description
`XC0079` (pg. 217)	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
`XC0201` (pg. 219)	It is a dynamic error if the `<p:uncompress>` step cannot perform the requested content-type cast.
`XC0202` (pg. 219)	It is a dynamic error if the compression format cannot be understood, determined and/or processed.
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.59 p:unwrap

Unwraps elements in a document.

Summary

```
<p:declare-step type="p:unwrap">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
</p:declare-step>
```

The `p:unwrap` step unwraps element nodes, specified by an XSLT selection pattern, from the document appearing on its `source` port. Unwrapping means: remove the matched element (including any attributes), but keep all its children.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to unwrap the matched elements in.

`result`	`output`	`true`	`text`
			`xml`
			`html`
				`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern for the nodes to unwrap, as a string. This must match element nodes (or the document-node). If any other kind of node is matched, error `XC0023` (pg. 156) is raised.

Description

The `p:unwrap` step takes the XSLT match pattern in the `match` option and holds this against the document appearing on its `source` port. This pattern must match element nodes (or the document-node). Matching elements are unwrapped: removed (including any attributes) but their child nodes remain. The resulting document appear on the `result` port.

If you want to remove an element including child nodes you need `p:delete` (pg. 36).

Examples

Basic usage

The following example unwraps all `<name>` elements. Note that nested `<name>` elements are also unwrapped.
Source document:

```
<person>
   <name>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
      <spouse>
         <name>
            <firstname>Clara</firstname>
            <lastname>Doe</lastname>
         </name>
      </spouse>
   </name>
</person>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:unwrap match="name"/>

</p:declare-step>
```

Result document:

```
<person>
   <firstname>John</firstname>
   <lastname>Doe</lastname>
   <spouse>
      <firstname>Clara</firstname>
      <lastname>Doe</lastname>
   </spouse>
</person>
```

Additional details

- In most cases, `p:unwrap` preserves all document-properties of the document appearing on its `source` port.
- For documents consisting of just a root-element containing text: if this root-element is unwrapped, the result is a document-node with a single text node child. This changes the result document's content-type (its `content-type` property) to `text/plain`. The `serialization` document-property, if present, is removed.
- If a document consisting of only an empty root-element is unwrapped, the result will be a document-node without any children. The result document's content type does not change if you do this.
- If you unwrap the document-node (set the `match` property to `/`) nothing will happen.
- It's possible to produce non well-formed XML using this step. Unwrapping the root-element from the following document (`<p:unwrap match="/*"/>` or simply `<p:unwrap/>`) produces a result with just a single comment node, which is not well-formed:

  ```
  <!-- Some comment -->
  <root/>
  ```

  XProc does not care about this, it keeps calm and carries on. However, writing this result to disk might not be what you expect.

Errors raised

Error code	Description
XC0023 (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.

2.60 p:uuid

Injects UUIDs into a document.

Summary

```
<p:declare-step type="p:uuid">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="text xml html" sequence="false"/>
  <option name="match" as="xs:string" required="false" select="'/*'"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="version" as="xs:integer?" required="false" select="()"/>
</p:declare-step>
```

The `p:uuid` step takes the document appearing on its `source` port and replaces nodes matching the `match` option with a UUID.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to inject the UUIDs in.

`result`	`output`	`true`	`text`
			`xml`
			`html`
				`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`match`	`xs:string` (XSLT selection pattern)	`false`	`/*`	The XSLT match pattern for the nodes to replace with the UUID.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the UUID generation. The XProc specification does not define any parameters for this option. A specific XProc processor might define its own.
`version`	`xs:integer?`	`false`	`()`	The UUID version to use for computing its value. Its default value is implementation-defined and therefore dependent on the XProc processor used. Version 4 UUIDs are always supported. Whether any other versions are supported is also implementation-defined.

Description

UUID stands for Universally Unique Identifier. It is also known as GUID, which stands for Globally Unique Identifier. A UUID is a 128-bit value that is, for all practical purposes, worldwide unique. It is usually written as a 32-character hexadecimal value in a pattern using hyphens, for example `f81d4fae-7dec-11d0-a765-00a0c91e6bf6`. The Wikipedia page about UUIDs is here (https://en.wikipedia.org/wiki/Universally_unique_identifier).

The `p:uuid` step does the following:

- It computes a *single* UUID, using the `version` and `parameters` options. This UUID is used for all replacements (so all replacements get the *same* value).
- It takes the document appearing on its `source` port and holds the XSLT selection pattern in the `match` option against this.
- For all matched nodes:
  - If the matched node is an attribute, the *value* of the attribute is replaced with the UUID.
  - If the document-node is matched, the full document will be replaced by UUID (and the result will therefore be a text document).
  - In all other cases, the full node is replaced by the UUID.

Examples

Basic usage

The following example replaces the text inside the `<uuid>` elements with a generated UUID:

Source document:

```
<thing>
   <uuid>UUID</uuid>
   <uuid>UUID</uuid>
</thing>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:uuid match="/thing/uuid/text()"/>

</p:declare-step>
```

Result document:

```
<thing>
   <uuid>b0886cca-8c42-41c4-af40-5f83a868c1ef</uuid>
   <uuid>b0886cca-8c42-41c4-af40-5f83a868c1ef</uuid>
</thing>
```

Please notice that the UUIDs are identical, the same value is used for every replacement.

If the `match` option matches an attribute, the *value* of the attribute is replaced:

Source document:

```
<thing uuid=""/>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:uuid match="/thing/@uuid"/>

</p:declare-step>
```

Result document:

```
<thing uuid="68f8620d-c061-4d2b-b6ee-163dc451593a"/>
```

Additional details

- `p:uuid` preserves all document-properties of the document(s) appearing on its `source` port.
  There is one exception: if the resulting document contains only text, the `content-type` document-property is changed to `text/plain` and the `serialization` document-property is removed.
- If an attribute called `xml:base` is added or changed, the base URI of the element is updated accordingly. See also category Base URI related (pg. 212).

Errors raised

Error code	Description
`XC0060` (pg. 216)	It is a dynamic error if the processor does not support the specified `version` of the UUID algorithm.

2.61 p:validate-with-dtd

Validates a document using a DTD.

Summary

```
<p:declare-step type="p:validate-with-dtd">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="report-format" as="xs:string" required="false" select="'xvrl'"/>
  <option name="serialization" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:validate-with-dtd` step validates the document appearing on the `source` port using DTD (https://en.wikipedia.org/wiki/Document_type_definition) (Document Type Definition) validation. The `result` port emits a copy of the source document, possibly augmented.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to validate.
`result`	`output`	`true`	`xml` `html`	`false`	A copy of the document that appeared on the `source` port. If validation was successful, the output may have been augmented by the DTD. (For example, default attributes may have been added).
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0210` (pg. 160) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0210` (pg. 160) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`report-format`	`xs:string`	`false`	`xvrl`	The format for the document on the `report` port. The value `xvrl` (default) will always work: the report will be in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.
`serialization`	`map(xs:QName, item()*)?`	`false`	`()`	This option can supply a map with serialization properties (https://www.w3.org/TR/xslt-xquery-serialization-31/) for serializing the document on the `source` port, before it is re-parsed for validation (see the description for an explanation). If the source document has a `serialization` document-property, the two sets of serialization properties are merged (properties in the document-property have precedence).

Description

The `p:validate-with-dtd` step validates the document appearing on the `source` port using DTD (https://en.wikipedia.org/wiki/Document_type_definition) (Document Type Definition) validation. This works a little differently than the other validation techniques: validation takes place by first serializing the document (as if written to disk) and subsequently re-parse it using a validating XML parser. The DTD (or a link to it) must be supplied by the source document itself or by the serialization process.

The serialization options (whether provide by the `serialization` document-property or the `serialization` option) *must* include at least a `doctype-system` property. Without a system identifier, the document cannot be successfully parsed with a validating parser.

Examples

Basic usage (valid source document)

Assume we have an input document, called `input-valid.xml`, that looks like this:

```
<address>
  <first>Douglas</first>
  <last>Adams</last>
  <phone>42</phone>
</address>
```

We want to validate this document using the following DTD, called `example.dtd`:

```
<!ELEMENT address (first, last, phone)>
<!ATTLIST address type CDATA #IMPLIED>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

To perform this validation using the `p:validate-with-dtd` step, we need to link the DTD to the document using the `doctype-system` serialization property. The output of the example is what is returned on the `report` port.

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-dtd serialization="map{'doctype-system' : 'example.dtd'}" name="validate"/>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.69+02:00</timestamp>
      <document href="file:/…/…/input-valid.xml"/>
      <validator name="org.apache.xerces.jaxp.SAXParserImpl$JAXPSAXParser"/>
   </metadata>
</report>
```

Basic usage (invalid source document)

Using the same DTD as in Basic usage (valid source document) (pg. 159), we're now going to validate an *in*valid document (called `input-invalid.xml`). Since we want to have a look at what comes out of the `report` port, we have to set the `assert-valid` option to `false`.

```
<address>
  <FIRST>Douglas</FIRST>
  <last>Adams</last>
  <phone>42</phone>
</address>
```

Performing this validation using the `p:validate-with-dtd` step returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-dtd assert-valid="false" serialization="map{'doctype-
system' : 'example.dtd'}" name="validate"/>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.94+02:00</timestamp>
      <document href="file:/…/…/input-invalid.xml"/>
      <validator name="org.apache.xerces.jaxp.SAXParserImpl$JAXPSAXParser"/>
   </metadata>
   <detection severity="fatal-error">
      <message>SAXParseException: Element type "FIRST" must be declared.</message>
   </detection>
</report>
```

Additional details

- If validation fails (and `assert-valid` is `false`), all document-properties on the `source` port are preserved on the `result` port. If validation succeeds, only the `base-uri` and `serialization` document-properties are preserved, the `content-type` document-property will be `application/xml`.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0210` (pg. 220)	It is a dynamic error if the `assert-valid` option on `<p:validate-with-dtd>` is `true` and the input document is not valid.

2.62 p:validate-with-json-schema

Validates a JSON document using JSON schema.

Summary

```
<p:declare-step type="p:validate-with-json-schema">
  <input port="source" primary="true" content-types="json" sequence="false"/>
  <output port="result" primary="true" content-types="json" sequence="false"/>
  <input port="schema" primary="false" content-types="json" sequence="false"/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="default-version" as="xs:string?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="report-format" as="xs:string" required="false" select="'xvrl'"/>
</p:declare-step>
```

The `p:validate-with-json-schema` step validates the JSON document appearing on the `source` port using JSON Schema (https://json-schema.org/draft/2020-12/json-schema-validation) validation. The JSON schema (a JSON document itself) is supplied through the `schema` port. The `result` port emits a copy of the source document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`json`	`false`	The document to validate.
`result`	`output`	`true`	`json`	`false`	A verbatim copy of the document that appeared on the `source` port.
`schema`	`input`	`false`	`json`	`false`	The JSON schema to validate against.
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0165` (pg. 163) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0165` (pg. 163) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`default-version`	`xs:string?`	`false`	`()`	Specifies the schema version if the schema itself doesn't specify one itself. If both the schema doesn't specify a version and this option is empty, the schema version is implementation-defined and therefore dependent on the XProc processor used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the validation. See "Validation parameters" on page 162 for more information.
`report-format`	`xs:string`	`false`	`xvrl`	The format for the document on the `report` port. The value `xvrl` (default) will always work: the report will be in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.

Description

The `p:validate-with-json-schema` step applies JSON Schema (https://json-schema.org/draft/2020-12/json-schema-validation)validation to the JSON document appearing on the `source` port. The JSON schema is supplied using the `schema` port. The outcome of the step, what appears on the `result` port, is a verbatim copy of the source document.

Validation parameters

The `p:validate-with-json-schema` step has a `parameters` port of datatype `map(xs:QName, item()*)?`. This (optional) map passes additional parameters for the validation process to the step:

- The parameters in this map, their values and semantics are implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:compile` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved for parameters for the schema *compilation* (if applicable). The value of this key must be a map itself.
- If the `report-format` option is set to `xvrl` (default): Any entries with keys in the `xvrl` namespace (`http://www.xproc.org/ns/xvrl`) are passed as parameters to the process that generates the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port. All standard XVRL generation parameters (https://spec.xproc.org/master/head/xvrl/#xvrl-generation) are supported.

Examples

Basic usage (valid source document)

Assume we have a JSON input document, called `input-valid.json`, that looks like this:

```
{
  "first_name": "Jane",
  "last_name": "Doe"
}
```

A JSON schema to validate this is as follows:

```
{
  "$id": "https://example.com/schemas/customer",
  "type": "object",
  "properties": {
    "first_name": {"type": "string"},
    "last_name": {"type": "string"},
    "email": {"type": "string"}
  },
  "required": [
    "first_name",
    "last_name"
  ]
}
```

Performing this validation using the `p:validate-with-json-schema` step returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-json-schema name="validate">
    <p:with-input port="schema" href="example-json-schema.json"/>
  </p:validate-with-json-schema>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.17+02:00</timestamp>
      <document href="file:/…/…/input-valid.json"/>
      <schema href="file:/…/…/example-json-schema.json" schematypens="JsonSchema"/>
      <validator name="networknt/json-schema-validator"/>
   </metadata>
   <digest fatal-error-count="0"
           error-count="0"
           warning-count="0"
           info-count="0"
           valid="true"/>
</report>
```

Basic usage (invalid source document)

Using the same JSON schema as in Basic usage (valid source document) (pg. 162), we're now going to validate an *in*valid document (called `input-invalid.json`). Since we want to have a look at what comes out of the `report` port, we have to set the `assert-valid` option to `false`.

```
{"last_name": "Doe"}
```

Performing this validation using the `p:validate-with-json-schema` step returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-json-schema assert-valid="false" name="validate">
    <p:with-input port="schema" href="example-json-schema.json"/>
  </p:validate-with-json-schema>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.41+02:00</timestamp>
      <document href="file:/…/…/input-invalid.json"/>
      <schema href="file:/…/…/example-json-schema.json" schematypens="JsonSchema"/>
      <validator name="networknt/json-schema-validator"/>
   </metadata>
   <detection severity="error" code="1028">
      <location jsonpath="$"/>
      <message>$.first_name: is missing but it is required</message>
   </detection>
   <digest fatal-error-count="0"
           error-count="1"
           warning-count="0"
           info-count="0"
           valid="false"/>
</report>
```

Additional details

- `p:validate-with-json-schema` preserves all document-properties of the document appearing on its `source` port for the document on its `result` port.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0117` (pg. 218)	It is a dynamic error if a report-format option was specified that the processor does not support.
`XC0163` (pg. 219)	It is a dynamic error if the selected version is not supported.
`XC0164` (pg. 219)	It is a dynamic error if the document supplied on `schema` port is not a valid JSON schema document in the selected version.
`XC0165` (pg. 219)	It is a dynamic error if the `assert-valid` option on `<p:validate-with-json-schema>` is `true` and the input document is not valid.

2.63 p:validate-with-nvdl

Validate a document using NVDL.

Summary

```
<p:declare-step type="p:validate-with-nvdl">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <input port="nvdl" primary="false" content-types="xml" sequence="false"/>
  <input port="schemas" primary="false" content-types="text xml" sequence="true">
    <p:empty/>
  <input/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="report-format" as="xs:string" required="false" select="'xvrl'"/>
</p:declare-step>
```

The `p:validate-with-nvdl` step validates the document appearing on the `source` port using NVDL (https://en.wikipedia.org/wiki/Namespace-based_Validation_Dispatching_Language) (Namespace-based Validation Dispatching Language) validation. The NVDL schema is supplied through the `nvdl` port. The `result` port emits a copy of the source document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to validate.
`result`	`output`	`true`	`xml` `html`	`false`	A verbatim copy of the document that appeared on the `source` port.
`nvdl`	`input`	`false`	`xml`	`false`	The NVDL schema to validate against.
`schemas`	`input`	`false`	`text` `xml`	`true`	Optional schemas, referenced from the NVDL schema by URI. See the description below for more information.
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0053` (pg. 167) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0053` (pg. 167) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the validation. See "Validation parameters" on page 165 for more information.
`report-format`	`xs:string`	`false`	`xvrl`	The format for the document on the `report` port. The value `xvrl` (default) will always work: the report will be in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.

Description

The `p:validate-with-nvdl` step applies NVDL (https://en.wikipedia.org/wiki/Namespace-based_Validation_Dispatching_Language) (Namespace-based Validation Dispatching Language) validation to the document appearing on the `source` port. The NVDL schema is supplied using the `nvdl` port. The outcome of the step, what appears on the `result` port, is a verbatim copy of the source document.

An NVDL schema usually refers to other schemas by URI. To find such a schema, the step first looks at the schemas provided on the `schema` port (if any). If a schema with the same base URI as mentioned in the NVDL schema is present on the `schemas` port, this is used. If not, the XProc processor will attempt to load it by its URI, usually from disk.

Validation parameters

The `p:validate-with-nvdl` step has a `parameters` port of datatype `map(xs:QName, item()*)?`. This (optional) map passes additional parameters for the validation process to the step:

- The parameters in this map, their values and semantics are implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:compile` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved for parameters for the schema *compilation* (if applicable). The value of this key must be a map itself.
- If the `report-format` option is set to `xvrl` (default): Any entries with keys in the `xvrl` namespace (`http://www.xproc.org/ns/xvrl`) are passed as parameters to the process that generates the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port. All standard XVRL generation parameters (https://spec.xproc.org/master/head/xvrl/#xvrl-generation) are supported.

Examples

Basic usage (valid source document)

Assume we have an input document, called `input-valid.xml`, that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<things xmlns:thingies="https://www.xprocref.org/ns/thingies">
  <thingies:thing id="A">A thing...</thingies:thing>
  <thingies:thing id="B">Another thing...</thingies:thing>
</things>
```

Since this document mixes namespaces, we want to validate it using NVDL. An NVDL schema for this, called `example.nvdl`, is as follows:

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="https://www.xprocref.org/ns/thingies">
    <validate schema="thingies.xsd"/>
  </namespace>
  <anyNamespace>
    <allow/>
  </anyNamespace>
</rules>
```

The `<thing>` elements in the `https://www.xprocref.org/ns/thingies` namespace are validated using the following simple XML schema called `thingies.xsd`. It says that the only thing allowed is a `<thing>` element with a required `id` attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/
XMLSchema" elementFormDefault="qualified" targetNamespace="https://www.xprocref.org/ns/thingies"
  xmlns="https://www.xprocref.org/ns/thingies">
  <xs:element name="thing">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="id" type="xs:NCName" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Performing this validation using the `p:validate-with-nvdl` step returns the following on the `report` port:
Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-nvdl name="validate">
    <p:with-input port="nvdl" href="example.nvdl"/>
  </p:validate-with-nvdl>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.72+02:00</timestamp>
      <document href="file:/…/…/input-valid.xml"/>
      <schema href="file:/…/…/example.nvdl"
              schematypens="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"/>
      <validator name="jing"/>
   </metadata>
   <digest fatal-error-count="0"
           error-count="0"
           warning-count="0"
           info-count="0"
           valid="true"/>
</report>
```

Basic usage (invalid source document)

Using the same NVDL schema as in Basic usage (valid source document) (pg. 165), we're now going to validate an *in*valid document (called `input-invalid.xml`). Since we want to have a look at what comes out of the `report` port, we have to set the `assert-valid` option to `false`.

```
<?xml version="1.0" encoding="UTF-8"?>
<things xmlns:thingies="https://www.xprocref.org/ns/thingies">
  <thingies:thing id="A" invalid-attribute="true">A thing...</thingies:thing>
  <thingies:thing id="B">Another thing...</thingies:thing>
</things>
```

Performing this validation using the `p:validate-with-nvdl` step returns the following on the `report` port:
Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-nvdl name="validate" assert-valid="false">
    <p:with-input port="nvdl" href="example.nvdl"/>
  </p:validate-with-nvdl>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.02+02:00</timestamp>
      <document href="file:/…/…/input-invalid.xml"/>
      <schema href="file:/…/…/example.nvdl"
              schematypens="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"/>
      <validator name="jing"/>
   </metadata>
   <detection severity="error">
      <location line="2" column="51"/>
      <message>cvc-complex-type.3.2.2: Attribute 'invalid-
attribute' is not allowed to appear in element 'thingies:thing'.</message>
   </detection>
   <digest fatal-error-count="0"
           error-count="1"
           warning-count="0"
           info-count="0"
           valid="false"/>
</report>
```

Additional details

- `p:validate-with-nvdl` preserves all document-properties of the document appearing on its `source` port for the document on its `result` port.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- The document appearing on the `result` port may have been enriched with PSVI (Post-Schema-Validation-Infoset) annotations (see the XML Schema recommendation (https://www.w3.org/TR/xmlschema-1/)).

Errors raised

Error code	Description
`XC0053` (pg. 216)	It is a dynamic error if the `assert-valid` option on `<p:validate-with-nvdl>` is `true` and the input document is not valid.
`XC0117` (pg. 218)	It is a dynamic error if a report-format option was specified that the processor does not support.
`XC0154` (pg. 219)	It is a dynamic error if the document supplied on `nvdl` port is not a valid NVDL document.

2.64 p:validate-with-relax-ng

Validate a document using RELAX NG.

Summary

```
<p:declare-step type="p:validate-with-relax-ng">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <input port="schema" primary="false" content-types="text xml" sequence="false"/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="dtd-attribute-values" as="xs:boolean" required="false" select="false()"/>
  <option name="dtd-id-idref-warnings" as="xs:boolean" required="false" select="false()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="report-format" as="xs:string" required="false" select="'xvrl'"/>
</p:declare-step>
```

The `p:validate-with-relax-ng` step validates the document appearing on the `source` port using RELAX NG (https://en.wikipedia.org/wiki/RELAX_NG) (REgular LAnguage for XML Next Generation) validation. The RELAX NG schema is supplied through the `schema` port. The `result` port emits a copy of the source document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to validate.
`result`	`output`	`true`	`xml` `html`	`false`	A verbatim copy of the document that appeared on the `source` port.
`schema`	`input`	`false`	`text` `xml`	`false`	The RELAX NG schema to validate against: • If the document appearing on this port is XML (has an XML media type), it must be a valid RELAX NG XML Syntax (https://en.wikipedia.org/wiki/RELAX_NG#XML_syntax) schema. • If the document appearing on this port is text (has a text media type), it must be a valid RELAX NG Compact Syntax (https://en.wikipedia.org/wiki/RELAX_NG#Compact_syntax) schema.
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0155` (pg. 171) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0155` (pg. 171) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`dtd-attribute-values`	`xs:boolean`	`false`	`false`	If `true`, the attribute value defaulting conventions of RELAX NG are applied. See the RELAX NG DTD Compatibility specification (https://www.oasis-open.org/committees/relax-ng/compatibility-20011203.html) for more information.
`dtd-id-idref-warnings`	`xs:boolean`	`false`	`false`	If `true`, a schema that is incompatible with the ID/IDREF/IDREFs feature of RELAX'NG DTD Compatibility as invalid. See the RELAX NG DTD Compatibility specification (https://www.oasis-open.org/committees/relax-ng/compatibility-20011203.html) for more information.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the validation. See "Validation parameters" on page 169 for more information.
`report-format`	`xs:string`	`false`	`xvrl`	The format for the document on the `report` port. The value `xvrl` (default) will always work: the report will be in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.

Description

The `p:validate-with-relax-ng` step applies RELAX NG (https://en.wikipedia.org/wiki/RELAX_NG) (REgular LAnguage for XML Next Generation) validation to the document appearing on the `source` port. The RELAX NG schema is supplied using the `schema` port. The outcome of the step, what appears on the `result` port, is a verbatim copy of the source document.

RELAX NG has two syntaxes: An XML based syntax (https://en.wikipedia.org/wiki/RELAX_NG#XML_syntax) and a text based syntax (https://en.wikipedia.org/wiki/RELAX_NG#Compact_syntax). Both can be used.

Validation parameters

The `p:validate-with-relax-ng` step has a `parameters` port of datatype `map(xs:QName, item()*)?`. This (optional) map passes additional parameters for the validation process to the step:

- The parameters in this map, their values and semantics are implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:compile` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved for parameters for the schema *compilation* (if applicable). The value of this key must be a map itself.
- If the `report-format` option is set to `xvrl` (default): Any entries with keys in the `xvrl` namespace (`http://www.xproc.org/ns/xvrl`) are passed as parameters to the process that generates the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port. All standard XVRL generation parameters (https://spec.xproc.org/master/head/xvrl/#xvrl-generation) are supported.

Examples

Basic usage (valid source document)

Assume we have an input document, called `input-valid.xml`, that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<things>
  <thing>A thing...</thing>
  <thing>Another thing...</thing>
</things>
```

A RELAX NG schema to validate this is as follows:

```
<grammar ns="" xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="things">
      <oneOrMore>
        <element name="thing">
          <text/>
        </element>
      </oneOrMore>
    </element>
  </start>
</grammar>
```

Performing this validation using the `p:validate-with-relax-ng` step returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-relax-ng name="validate">
    <p:with-input port="schema" href="example.rng"/>
  </p:validate-with-relax-ng>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.3+02:00</timestamp>
      <document href="file:/…/…/input-valid.xml"/>
      <schema href="file:/…/…/example.rng"
              schematypens="http://relaxng.org/ns/structure/1.0"/>
      <validator name="jing"/>
   </metadata>
   <digest fatal-error-count="0"
           error-count="0"
           warning-count="0"
           info-count="0"
           valid="true"/>
</report>
```

Basic usage (invalid source document)

Using the same RELAX NG schema as in Basic usage (valid source document) (pg. 169), we're now going to validate an *in*valid document (called `input-invalid.xml`). Since we want to have a look at what comes out of the `report` port, we have to set the `assert-valid` option to `false`.

```
<?xml version="1.0" encoding="UTF-8"?>
<things xmlns:thingies="https://www.xprocref.org/ns/thingies">
  <thingies:thing id="A" invalid-attribute="true">A thing...</thingies:thing>
  <thingies:thing id="B">Another thing...</thingies:thing>
</things>
```

Performing this validation using the `p:validate-with-relax-ng` step returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-relax-ng assert-valid="false" name="validate">
    <p:with-input port="schema" href="example.rng"/>
  </p:validate-with-relax-ng>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.56+02:00</timestamp>
      <document href="file:/…/…/input-invalid.xml"/>
      <schema href="file:/…/…/example.rng"
              schematypens="http://relaxng.org/ns/structure/1.0"/>
      <validator name="jing"/>
   </metadata>
   <detection severity="error">
      <location line="3" column="16"/>
      <message>element "thing-error" not allowed anywhere; expected the element end-tag or element "thing"</
message>
   </detection>
   <digest fatal-error-count="0"
           error-count="1"
           warning-count="0"
           info-count="0"
           valid="false"/>
</report>
```

Additional details

- `p:validate-with-relax-ng` preserves all document-properties of the document appearing on its `source` port for the document on its `result` port.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- The document appearing on the `result` port may have been enriched with PSVI (Post-Schema-Validation-Infoset) annotations (see the XML Schema recommendation (https://www.w3.org/TR/xmlschema-1/)).

Errors raised

Error code	Description
`XC0117` (pg. 218)	It is a dynamic error if a report-format option was specified that the processor does not support.
`XC0153` (pg. 219)	It is a dynamic error if the document supplied on `schema` port cannot be interpreted as an RELAX NG Grammar.
`XC0155` (pg. 219)	It is a dynamic error if the `assert-valid` option on `<p:validate-with-relax-ng>` is `true` and the input document is not valid.

2.65 p:validate-with-schematron

Validates a document using Schematron.

Summary

```
<p:declare-step type="p:validate-with-schematron">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <input port="schema" primary="false" content-types="xml" sequence="false"/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="phase" as="xs:string" required="false" select="'#DEFAULT'"/>
  <option name="report-format" as="xs:string" required="false" select="'svrl'"/>
</p:declare-step>
```

The `p:validate-with-schematron` step validates the document appearing on the `source` port using Schematron validation. The Schematron schema is supplied through the `schema` port. The `result` port emits a copy of the source document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to validate.
`result`	`output`	`true`	`xml` `html`	`false`	A verbatim copy of the document that appeared on the `source` port.
`schema`	`input`	`false`	`xml`	`false`	The Schematron schema to validate against.
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0054` (pg. 176) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0054` (pg. 176) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters controlling the validation. See "Validation parameters" on page 172 for more information.
`phase`	`xs:string`	`false`	`#DEFAULT`	The Schematron schema phase to select.
`report-format`	`xs:string`	`false`	`svrl`	The format for the document on the `report` port: • The value `svrl` (default) produces a report in SVRL (https://schematron.com/document/3427.html) (Schematron Validation Report Language). • The value `xvrl` (default) produces a report in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). • Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.

Description

The `p:validate-with-schematron` step applies Schematron (https://schematron.com/) validation to the document appearing on the `source` port. The Schematron schema must be supplied using the `schema` port. The outcome of the step, what appears on the `result` port, is a verbatim copy of the source document.

Validation parameters

The `p:validate-with-schematron` step has a `parameters` port of datatype `map(xs:QName, item()*)?`. This (optional) map passes additional parameters for the validation process to the step, which correspond to Schematron external variables, to parameters that influence code generation, or to parameters that influence SVRL to XVRL conversion.

- The parameters in this map, their values and semantics are implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:implementation` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved to select a Schematron implementation the XProc processor supports.The list of supported Schematron implementations and their associated values is implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:compile` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved for parameters for the schema *compilation* (if applicable). The value of this key must be a map itself.

  For instance, if a code-generating implementation such as SchXslt (https://github.com/schxslt/schxslt) is used, the entries of the `c:compile` map are passed to the code generator.
- If the `report-format` option is set to `xvrl` (default): Any entries with keys in the `xvrl` namespace (`http://www.xproc.org/ns/xvrl`) are passed as parameters to the process that generates the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port. All standard XVRL generation parameters (https://spec.xproc.org/master/head/xvrl/#xvrl-generation) are supported.

Examples

Basic usage (valid source document)

Assume we have an input document, called `input-valid.xml`, that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<things>
  <thing id="A">A thing...</thing>
  <thing id="B">Another thing... referencing <thingref idref="A"/>!</thing>
</things>
```

Any `<thingref>` elements must reference existing `<thing>` elements by identifier. Here is a simple Schematron schema that validates this:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">

  <pattern>
    <rule context="thingref/@idref">
      <let name="id" value="string(.)"/>
      <assert test="exists(//thing[@id eq $id])">Reference to non-existent id: "<value-of select="$id"/>"</
assert>
    </rule>
  </pattern>

</schema>
```

Performing this validation using the `p:validate-with-schematron` step (which, for this example, uses the SchXslt (https://github.com/schxslt/schxslt) Schematron processor) returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-schematron name="validate">
    <p:with-input port="schema" href="example.sch"/>
  </p:validate-with-schematron>

</p:declare-step>
```

Result document:

```
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl" phase="#ALL">
   <svrl:active-pattern documents="file:/…/…/input-valid.xml?xproc_uniqueid=0"/>
   <svrl:fired-rule context="thingref/@idref"/>
</svrl:schematron-output>
```

The same example, but now producing a XVRL (https://spec.xproc.org/master/head/xvrl/) format report, is as follows:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-schematron name="validate" report-format="xvrl">
    <p:with-input port="schema" href="example.sch"/>
  </p:validate-with-schematron>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.3+02:00</timestamp>
      <document href="file:/…/…/input-valid.xml"/>
      <schema href="file:/…/…/example.sch"
              schematypens="http://purl.oclc.org/dsdl/schematron"/>
      <validator name="SchXslt2"/>
   </metadata>
   <digest fatal-error-count="0"
           error-count="0"
           warning-count="0"
           info-count="0"
           valid="true"/>
</report>
```

The exact format of the reports might differ across implementations. Please experiment before using it.

Basic usage (invalid source document)

Using the same Schematron schema as in Basic usage (valid source document) (pg. 173), we're now going to validate an *in*valid document (called `input-invalid.xml`). Since we want to have a look at what comes out of the `report` port, we have to set the `assert-valid` option to `false`.

```
<?xml version="1.0" encoding="UTF-8"?>
<things>
  <thing id="A">A thing...</thing>
  <thing id="B">Another thing... referencing <thingref idref="C"/>!</thing>
</things>
```

Performing this validation using the `p:validate-with-schematron` step (which, for this example, uses the SchXslt (https://github.com/schxslt/schxslt) Schematron processor) returns the following on the `report` port:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-schematron assert-valid="false" name="validate">
    <p:with-input port="schema" href="example.sch"/>
  </p:validate-with-schematron>

</p:declare-step>
```

Result document:

```
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl" phase="#ALL">
   <svrl:active-pattern documents="file:/…/…/input-invalid.xml?xproc_uniqueid=0"/>
   <svrl:fired-rule context="thingref/@idref"/>
   <svrl:failed-assert test="exists(//thing[@id eq $id])"
                       location="/Q{}things[1]/Q{}thing[2]/Q{}thingref[1]/@idref">
      <svrl:text>Reference to non-existent id: "C"</svrl:text>
   </svrl:failed-assert>
</svrl:schematron-output>
```

The same example, but now producing a XVRL (https://spec.xproc.org/master/head/xvrl/) format report, is as follows:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-schematron name="validate" assert-valid="false" report-format="xvrl">
    <p:with-input port="schema" href="example.sch"/>
  </p:validate-with-schematron>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.08+02:00</timestamp>
      <document href="file:/…/…/input-invalid.xml"/>
      <schema href="file:/…/…/example.sch"
              schematypens="http://purl.oclc.org/dsdl/schematron"/>
      <validator name="SchXslt2"/>
   </metadata>
   <detection severity="error">
      <location xpath="/Q{}things[1]/Q{}thing[2]/Q{}thingref[1]/@idref"/>
      <message>Reference to non-existent id: "C"</message>
   </detection>
   <digest fatal-error-count="0"
           error-count="1"
           warning-count="0"
           info-count="0"
           valid="false"/>
</report>
```

Again, the exact format of the reports might differ across implementations. Please experiment before using it.

Another way of handling validation errors is to have `p:validate-with-schematron` raise its error `XC0054` (pg. 176) (by setting the `assert-valid` option to `true`) and catch this in a `<p:try>`/`<p:catch>` construction. The following pipeline shows you the `<c:errors>` result, that is available inside the `<p:catch>`:

```
<p:declare-step xmlns:err="http://www.w3.org/ns/xproc-error" xmlns:p="http://www.w3.org/ns/
xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:try>
    <p:validate-with-schematron assert-valid="true">
      <p:with-input port="schema" href="example.sch"/>
    </p:validate-with-schematron>
    <p:catch code="err:XC0054">
      <p:identity/>
    </p:catch>
  </p:try>

</p:declare-step>
```

Result document:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:error xmlns:err="http://www.w3.org/ns/xproc-error"
            code="err:XC0054"
            name="!1.1.1.1"
            type="p:validate-with-schematron"
            href="file:/…/…/…"
            line="7"
            column="53">
      <svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl" phase="#ALL">
         <svrl:active-pattern documents="file:/…/…/input-invalid.xml?xproc_uniqueid=0"/>
         <svrl:fired-rule context="thingref/@idref"/>
         <svrl:failed-assert test="exists(//thing[@id eq $id])"
                             location="/Q{}things[1]/Q{}thing[2]/Q{}thingref[1]/@idref">
            <svrl:text>Reference to non-existent id: "C"</svrl:text>
         </svrl:failed-assert>
      </svrl:schematron-output>
   </c:error>
</c:errors>
```

The exact contents of the `<c:errors>` element might differ across implementations. Please experiment before using it.

Additional details

- `p:validate-with-schematron` preserves all document-properties of the document appearing on its `source` port for the document on its `result` port.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- The document appearing on the `result` port may have been enriched with PSVI (Post-Schema-Validation-Infoset) annotations (see the XML Schema recommendation (https://www.w3.org/TR/xmlschema-1/)).

Errors raised

Error code	Description
`XC0054` (pg. 216)	It is a dynamic error if the `assert-valid` option is `true` and any Schematron assertions fail.
`XC0117` (pg. 218)	It is a dynamic error if a report-format option was specified that the processor does not support.
`XC0151` (pg. 219)	It is a dynamic error if the document supplied on `schema` port is not a valid Schematron document.

2.66 p:validate-with-xml-schema

Validates a document using XML Schema.

Summary

```
<p:declare-step type="p:validate-with-xml-schema">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="xml html" sequence="false"/>
  <input port="schema" primary="false" content-types="xml" sequence="true"/>
  <output port="report" primary="false" content-types="xml json" sequence="true"/>
  <option name="assert-valid" as="xs:boolean" required="false" select="true()"/>
  <option name="mode" as="item()*" required="false" select="'strict'" values="('strict','lax')"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="report-format" as="xs:string" required="false" select="'xvrl'"/>
  <option name="try-namespaces" as="xs:boolean" required="false" select="false()"/>
  <option name="use-location-hints" as="xs:boolean" required="false" select="false()"/>
  <option name="version" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:validate-with-xml-schema` step validates the document appearing on the `source` port using XML Schema validation. The most common way to provide a schema is through its `schema` port. The `result` port emits a copy of the source document with default attributes/elements filled in and (optional) PSVI annotations.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`false`	The document to validate.

`result`	`output`	`true`	`xml`
			`html`
				`false`	The document that appeared on the `source` port with the following alterations (see also the XML Schema recommendation (https://www.w3.org/TR/xmlschema-1/)):
					• If the XProc processor supports PSVI (Post-Schema-Validation-Infoset) annotations:
					  • The document is *valid*: the source document with PSVI annotations and any defaulting of attributes and elements filled in.
					  • The document is *invalid* and the `assert-valid` option is `false`: the source document with maybe some PSVI annotations (at least for the sub-trees that are valid).
					• If PSVI annotations are *not* supported by the XProc processor used:
					  • The document is *valid*: the source document with any defaulting of attributes and elements filled in.
					  • The document is *invalid* and the `assert-valid` option is `false`: the source document, unchanged.
					When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0156` (pg. 183) is raised.

`schema`	`input`	`false`	`xml`	`true`	Schema(s) to validate against. Providing a schema (or more than one) on this port is the most common way of supplying schemas to the step. There are other ways to provide schemas, see “Locating schemas” on page 179 for more information.
`report`	`output`	`false`	`xml` `json`	`true`	A report that describes the validation results, both for valid and invalid source documents. The format for this report is determined by the `report-format` option. When the `assert-valid` option is `true` and the document is *invalid*, nothing will appear on this port because error `XC0156` (pg. 183) is raised.

Options:

Name	Type	Req?	Default	Description
`assert-valid`	`xs:boolean`	`false`	`true`	Determines what happens if the document is *invalid*: • If `true`, error `XC0156` (pg. 183) is raised. • If `false`, the step always succeeds. The validity of the document must be determined by inspecting the document that appears on the `report` port.
`mode`	`item()*`	`false`	`strict`	This option controls how the schema validation starts: • Setting this to `strict` means that the document element must be declared and schema-valid, otherwise it will be treated as invalid. • Setting this to `lax` means that the absence of a declaration for the document element does not itself count as an unsuccessful outcome of validation. See Validating in lax mode (pg. 183) for an example.
`parameters`	`map(xs:QName, item()*)`	`false`	`()`	Parameters controlling the validation. See "Validation parameters" on page 179 for more information.
`report-format`	`xs:string`	`false`	`xvrl`	The format for the document on the `report` port. The value `xvrl` (default) will always work: the report will be in XVRL (https://spec.xproc.org/master/head/xvrl/) (Extensible Validation Report Language). Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor used.
`try-namespaces`	`xs:boolean`	`false`	`false`	Whether to try to dereference any namespace URIs in the source document for locating schemas. See "Locating schemas" on page 179 for more information.
`use-location-hints`	`xs:boolean`	`false`	`false`	Determines what to do with schema location hints in the source document. See "Locating schemas" on page 179 for more information.
`version`	`xs:string?`	`false`	`()`	If this option is set, the specified version of XML Schema must be used for validation. Likely values are `1.0` or `1.1`. Which XML Schema versions are supported is implementation-defined and therefore dependent on the XProc processor used. In all likelihood, version `1.0` will always be supported. If this option is *not* set, the XML schema version use and therefore dependent on the XProc processor used. For instance, it might be simply `1.0`, or the XProc processor might take a look at the XML schema itself to determine the version.

Description

The `p:validate-with-xml-schema` step validates the document appearing on the `source` against one or more W3C XML Schema(s) (https://www.w3.org/TR/xmlschema-1/).

The schema(s) used for validation can be provided in several ways. Probably the most common way is to provide them on the `schema` port. Another likely way to provide schemas is using schema references in the source document. If you want the `p:validate-with-xml-schema` step to do this, you must set the `use-location-hint` option to `true`. For more information about providing schemas see the "Locating schemas" on page 179 section below.

The outcome of the step, what appears on the `result` port, is a copy of the source document with a few alterations. If the document is valid all default attributes and elements will be filled in. If the processor supports PSVI annotations (as described in the XML Schema recommendation (https://www.w3.org/TR/xmlschema-1/)) these will be present to. For details see the description of the `result` port.

Locating schemas

One or more schemas can be provided on the `schema` port. But it is also possible the document on the `source` port contains schema references on its own, for instance an `xsi:schemaLocation` attribute. So which schema(s) should the step use for validation? The rules are as follows:

- If documents are provided on the `schema` port, these will be used. For most use-cases, this is the preferred way of providing the schema(s).
- If there are no schemas supplied on the `schema` port:
  - If the `use-location-hint` option is `true`, the XProc processor will have a look at schema references in the source document. Which location hints it will recognize as such is implementation-defined and therefore dependent on the XProc processor used. However, most probably, the `xsi:noNamespaceSchemaLocation` and `xsi:schemaLocation` attributes should do the trick (the `xsi` namespace prefix here is bound to the `http://www.w3.org/2001/XMLSchema-instance` namespace). See Using location hints (pg. 182) for an example.

    If the `use-location-hint` option is `false` (default), schema references in the source document are ignored.
  - If the `try-namespaces` option is `true`, the XProc processor will try to retrieve the schema for a namespace using the namespace URI. So if we have a document in the `http://www.something.org/ns/documents` namespace, the XProc processor will perform an HTTP GET request on this URI. If this returns a valid XML schema, the show is on. Some implementations might also be able to handle RDDL (https://rddl.org/) documents that refer to schemas.

    If the `try-namespaces` option is `false` (default) no attempt like this will be made.

Validation parameters

The `p:validate-with-xml-schema` step has a `parameters` port of datatype `map(xs:QName, item()*)?`. This (optional) map passes additional parameters for the validation process to the step:

- The parameters in this map, their values and semantics are implementation-defined and therefore dependent on the XProc processor used.
- A special entry with key `c:compile` (the `c` namespace prefix is bound to the standard XProc namespace `http://www.w3.org/ns/xproc-step`) is reserved for parameters for the schema *compilation* (if applicable). The value of this key must be a map itself.
- If the `report-format` option is set to `xvrl` (default): Any entries with keys in the `xvrl` namespace (`http://www.xproc.org/ns/xvrl`) are passed as parameters to the process that generates the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port. All standard XVRL generation parameters (https://spec.xproc.org/master/head/xvrl/#xvrl-generation) are supported.

Examples

Basic usage (valid source document)

We're going to use a schema, that validates simple XML documents, consisting of a `<things>` root element and zero or more `<thing>` children. The root element has an optional attribute called `status` with default value `normal`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="things">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="thing" type="xs:string"/>
      </xs:sequence>
      <xs:attribute default="normal" name="status" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Let's use this schema to validate a valid document (called `input-valid.xml`) and see what comes out of the `result` port:

```
<things>
   <thing>A thing...</thing>
   <thing>Another thing...</thing>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:validate-with-xml-schema>
    <p:with-input port="schema" href="example.xsd"/>
  </p:validate-with-xml-schema>

</p:declare-step>
```

Result document:

```
<things status="normal">
   <thing>A thing...</thing>
   <thing>Another thing...</thing>
</things>
```

Notice that the missing optional attribute `status`, as defined in the schema, has been added to the `<things>` root element, with its default value `normal`. This will happen to every optional attribute and/or element that is not present in the source.

Now let's have a look at the XVRL (https://spec.xproc.org/master/head/xvrl/) report appearing on the `report` port (for the same, valid, source document):

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-xml-schema name="validate">
    <p:with-input port="schema" href="example.xsd"/>
  </p:validate-with-xml-schema>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.99+02:00</timestamp>
      <document href="file:/…/…/input-valid.xml"/>
      <schema href="file:/…/…/example.xsd"
              schematypens="http://www.w3.org/2001/XMLSchema"/>
      <validator name="org.apache.xerces.jaxp.validation.XMLSchemaFactory"/>
   </metadata>
   <digest fatal-error-count="0"
           error-count="0"
           warning-count="0"
           info-count="0"
           valid="true"/>
</report>
```

The exact format of the report might differ across implementations. Please experiment before using it.

Basic usage (invalid source document)

We're going to use the same schema as in Basic usage (valid source document) (pg. 179), but now provide an *invalid* source document (called `input-invalid.xml`):

```
<things>
   <thing>A thing...</thing>
   <thing-error>Another thing...</thing-error>
</things>
```

The pipeline will catch the resulting XVRL (https://spec.xproc.org/master/head/xvrl/) report. Please notice that we need to set the `assert-valid` option to `false`. If we had left it to its default value `true`, error `XC0156` (pg. 183) would have been raised.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="report@validate"/>

  <p:validate-with-xml-schema assert-valid="false" name="validate">
    <p:with-input port="schema" href="example.xsd"/>
  </p:validate-with-xml-schema>

</p:declare-step>
```

Result document:

```
<report xmlns="http://www.xproc.org/ns/xvrl">
   <metadata>
      <timestamp>2025-06-25T13:24:55.3+02:00</timestamp>
      <document href="file:/…/…/input-invalid.xml"/>
      <schema href="file:/…/…/example.xsd"
              schematypens="http://www.w3.org/2001/XMLSchema"/>
      <validator name="org.apache.xerces.jaxp.validation.XMLSchemaFactory"/>
   </metadata>
   <detection severity="error">
      <location line="3" xpath="/Q{}things[1]/Q{}thing-error[1]"/>
      <message>cvc-complex-type.2.4.a: Invalid content was found starting with element 'thing-
error'. One of '{thing}' is expected.</message>
   </detection>
   <digest fatal-error-count="0"
           error-count="1"
           warning-count="0"
           info-count="0"
           valid="false"/>
</report>
```

Again, the exact format of the report might differ across implementations. Please experiment before using it.

Another way of handling validation errors is to have `p:validate-with-xml-schema` raise its error `XC0156` (pg. 183) and catch this in a `<p:try>`/`<p:catch>` construction. The following pipeline shows you the `<c:errors>` result, that is available inside the `<p:catch>`:

```
<p:declare-step xmlns:err="http://www.w3.org/ns/xproc-error" xmlns:p="http://www.w3.org/ns/
xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:try>
    <p:validate-with-xml-schema>
      <p:with-input port="schema" href="example.xsd"/>
    </p:validate-with-xml-schema>
    <p:catch code="err:XC0156">
      <p:identity/>
    </p:catch>
  </p:try>

</p:declare-step>
```

Result document:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step">
   <c:error xmlns:err="http://www.w3.org/ns/xproc-error"
            code="err:XC0156"
            name="!1.1.1.1"
            type="p:validate-with-xml-schema"
            href="file:/…/…/…"
            line="7"
            column="33">
      <report xmlns="http://www.xproc.org/ns/xvrl">
         <metadata>
            <timestamp>2025-06-25T13:24:55.58+02:00</timestamp>
            <document href="file:/…/…/input-invalid.xml"/>
            <schema href="file:/…/…/example.xsd"
                    schematypens="http://www.w3.org/2001/XMLSchema"/>
            <validator name="org.apache.xerces.jaxp.validation.XMLSchemaFactory"/>
         </metadata>
         <detection severity="error">
            <location line="3" xpath="/Q{}things[1]/Q{}thing-error[1]"/>
            <message>cvc-complex-type.2.4.a: Invalid content was found starting with element 'thing-
error'. One of '{thing}' is expected.</message>
         </detection>
         <digest fatal-error-count="0"
                 error-count="1"
                 warning-count="0"
                 info-count="0"
                 valid="false"/>
      </report>
   </c:error>
</c:errors>
```

The exact contents of the `<c:errors>` element might differ across implementations. Please experiment before using it.

Using location hints

Sometimes you have source documents that already contain schema references, for instance:

```
<things>
   <thing>A thing...</thing>
   <thing>Another thing...</thing>
</things>
```

If we want the `p:validate-with-xml-schema` step to use this reference, we have to set the `try-location-hints` to `true`. We don't need to validate against any other schemas, so we set the `schema` port to empty.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:validate-with-xml-schema>
    <p:with-input port="schema" href="example.xsd"/>
  </p:validate-with-xml-schema>

</p:declare-step>
```

Result document:

```
<things status="normal">
   <thing>A thing...</thing>
   <thing>Another thing...</thing>
</things>
```

Validating in lax mode

Usually you want a document to completely validate against a schema. However, there are use-cases where the documents to validate are *wrapped* inside some root element. This happens, for instance, when in XProc you have a sequence of documents and use `p:wrap-sequence` (pg. 187) to wrap these results into a single XML document. The `p:validate-with-xml-schema` step allows you to disregard the root element and validate its child elements only by setting the `mode` option to `lax`.

Source document:

```
<weird-root-element>
   <things>
      <thing>A thing...</thing>
      <thing>Another thing...</thing>
   </things>
   <things/>
</weird-root-element>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:validate-with-xml-schema mode="lax">
    <p:with-input port="schema" href="example.xsd"/>
  </p:validate-with-xml-schema>

</p:declare-step>
```

Result document:

```
<weird-root-element>
   <things status="normal">
      <thing>A thing...</thing>
      <thing>Another thing...</thing>
   </things>
   <things status="normal"/>
</weird-root-element>
```

Additional details

- `p:validate-with-xml-schema` preserves all document-properties of the document appearing on its `source` port for the document on its `result` port.
- The document appearing on the `report` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- A schema can contain `<xs:include>` or `<xs:import>` elements. It is implementation-defined, and therefore dependent on the XProc processor used, if the documents supplied on the `schema` port are considered when resolving these elements.

Errors raised

Error code	Description
`XC0011` (pg. 216)	It is a dynamic error if the specified schema version is not available.
`XC0055` (pg. 216)	It is a dynamic error if the implementation does not support the specified mode.
`XC0117` (pg. 218)	It is a dynamic error if a report-format option was specified that the processor does not support.
`XC0152` (pg. 219)	It is a dynamic error if the document supplied on `schema` port is not a valid XML schema document.
`XC0156` (pg. 219)	It is a dynamic error if the `assert-valid` option on `<p:validate-with-xml-schema>` is `true` and the input document is not valid.

2.67 p:wrap

Wraps nodes in a parent element.

Summary

```
<p:declare-step type="p:wrap">
  <input port="source" primary="true" content-types="xml html" sequence="false"/>
  <output port="result" primary="true" content-types="application/xml" sequence="false"/>
  <option name="match" as="xs:string" required="true"/>
  <option name="wrapper" as="xs:QName" required="true"/>
  <option name="attributes" as="map(xs:QName, xs:anyAtomicType)?" required="false" select="()"/>
  <option name="group-adjacent" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:wrap` step wraps matching nodes in the document on the `source` into a new parent element.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml html`	`false`	The document that contains the nodes to wrap.
`result`	`output`	`true`	`application/xml`	`false`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`match`	`xs:string` (XSLT selection pattern)	`true`		The XSLT match pattern for the nodes to wrap, as a string. This must match either the document-node, an element, a processing-instruction, or a comment. If any other kind of node is matched, error `XC0023` (pg. 187) is raised.
`wrapper`	`xs:QName`	`true`		The name of the wrapping element.
`attributes`	`map(xs:QName, xs:anyAtomicType)?`	`false`	`()`	An optional map with entries (attribute name, attribute value). The attributes specified in this map are created on the wrapper element. Specifying attributes using this option works the same as performing a `p:wrap` step (without an `attributes` option), directly followed by a `p:set-attributes` (pg. 124) step.
`group-adjacent`	`xs:string?` (XPath expression)	`false`	`()`	An XPath expression to use for grouping the wrapped elements. See "Grouping" on page 184 below.

Description

The `p:wrap` step takes the XSLT match pattern in the `match` option and holds this against the document appearing on its `source` port. This pattern must match the document-node, an element, a processing-instruction, or a comment. The matched node is wrapped in an element, as specified in the `wrapper` option.

The `p:wrap` step perform a "deep" wrapping: the children of any matched node are also processed. Wrappers are added to *all* matching nodes.

You can't use `p:wrap` to wrap a text document. For this use `p:wrap-sequence` (pg. 187).

Grouping

The `group-adjacent` option of `p:wrap` allows you to group adjacent matching nodes in a single wrapper element. The value of this option must a valid XPath expression. It is evaluated for every node in the source document. Adjacent nodes with an equal result value are wrapped together in the same wrapper element.

For all nodes in the document on the `source` port:

- The node becomes the context item (accessible with the dot `.` operator).
- The expression in the `group-adjacent` option is evaluated.

- Two values of sibling nodes, computed by the XPath expression, are considered equal if the XPath function `deep-equal()` (https://www.w3.org/TR/xpath-functions-31/#func-deep-equal) returns `true` for them. In most cases this simply means that values are equal when you intuitively expect them to be.
- All sibling nodes with equal values for the XPath expression, that are adjacent, are wrapped together in a wrapper element, as named by the `wrapper` option.

For an example see Grouping and wrapping (pg. 185).

Examples

Basic usage

The following example wraps every `thing` element in a `<computer-part>` element:

Source document:

```
<things>
   <thing name="laptop"/>
   <thing name="desktop">
      <subthings>
         <thing name="keyboard"/>
         <thing name="mouse"/>
      </subthings>
   </thing>
</things>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true"/>
  <p:output port="result" sequence="true"/>

  <p:wrap match="thing" wrapper="computer-part"/>

</p:declare-step>
```

Result document:

```
<things>
   <computer-part>
      <thing name="laptop"/>
   </computer-part>
   <computer-part>
      <thing name="desktop">
         <subthings>
            <computer-part>
               <thing name="keyboard"/>
            </computer-part>
            <computer-part>
               <thing name="mouse"/>
            </computer-part>
         </subthings>
      </thing>
   </computer-part>
</things>
```

Please notice that the nested `<thing>` elements (inside the `<subthings>` element) are also wrapped. This is because `p:wrap` performs “deep” wrapping.

Grouping and wrapping

This example shows what happens when you use the `group-adjacent` option. Here we group fruits by color:

Source document:

```
<fruits>
   <fruit name="banana" color="yellow"/>
   <fruit name="orange" color="orange"/>
   <fruit name="carrot" color="orange"/>
   <fruit name="lemon" color="yellow"/>
</fruits>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true"/>
  <p:output port="result" sequence="true"/>

  <p:wrap match="fruit" wrapper="fruits-by-color" group-adjacent="@color"/>

</p:declare-step>
```

Result document:

```
<fruits>
   <fruits-by-color>
      <fruit color="yellow" name="banana"/>
   </fruits-by-color>
   <fruits-by-color>
      <fruit color="orange" name="orange"/>
      <fruit color="orange" name="carrot"/>
   </fruits-by-color>
   <fruits-by-color>
      <fruit color="yellow" name="lemon"/>
   </fruits-by-color>
</fruits>
```

You might have expected that the result would group the fruits together by color, resulting in groups of elements, wrapped in `<fruits-by-color>` elements: one for `banana`+`lemon` and one for `orange`+`carrot`. But `p:wrap` groups sibling nodes that are *adjacent* to each other only. It does *not* do what would be called "group by": all sibling nodes with the same value for the XPath expression together in a single group. If you need this, you will have to sort the document first. But unfortunately, XProc does not have anything on board for that. For more complex grouping, the advice is to use XSLT or XQuery.

Replacing comments by elements

This example shows that you can use `p:wrap` not only to wrap elements, but also other kinds of nodes. Here we use its functionality, together with the `p:string-replace` (pg. 134) step, to change comments into elements. For this, we first wrap every element in a `<comment>` element and then turn these comments into text nodes using `p:string-replace` (pg. 134).

Source document:

```
<examples>
     <!--Comment 1-->
     <!--Another comment…-->
   </examples>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true"/>
  <p:output port="result" sequence="true"/>

  <p:wrap match="comment()" wrapper="comment"/>
  <p:string-replace match="comment/comment()" replace="string(.)"/>

</p:declare-step>
```

Result document:

```
<examples>
   <comment>Comment 1</comment>
   <comment>Another comment…</comment>
</examples>
```

Additional details

- `p:wrap` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0023` (pg. 216)	It is a dynamic error if the selection pattern matches a wrong type of node.

2.68 p:wrap-sequence

Wraps a sequence of documents in an element.

Summary

```
<p:declare-step type="p:wrap-sequence">
  <input port="source" primary="true" content-types="text xml html" sequence="true"/>
  <output port="result" primary="true" content-types="application/xml" sequence="true"/>
  <option name="wrapper" as="xs:QName" required="true"/>
  <option name="attributes" as="map(xs:QName, xs:anyAtomicType)?" required="false" select="()"/>
  <option name="group-adjacent" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:wrap-sequence` step takes a sequence of documents on its `source` port and wraps these in a wrapper element. The result appears on the `result` port. It can also group the source document(s) before wrapping, based on an XPath expression.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`text xml html`	`true`	The document(s) to wrap.
`result`	`output`	`true`	`application/xml`	`true`	The resulting wrapped document(s)

Options:

Name	Type	Req?	Default	Description
`wrapper`	`xs:QName`	`true`		The element to wrap the document(s) in.
`attributes`	`map(xs:QName, xs:anyAtomicType)?`	`false`	`()`	An optional map with entries (attribute name, attribute value). The attributes specified in this map are created on the wrapper element. Specifying attributes using this option works the same as performing a `p:wrap-sequence` step (without an `attributes` option), directly followed by a `p:set-attributes` (pg. 124) step.
`group-adjacent`	`xs:string?` (XPath expression)	`false`	`()`	An XPath expression to use for grouping the wrapped elements. See "Grouping" on page 187 below.

Description

Basic usage of the `p:wrap-sequence` step is to wrap an element around a sequence of text, XML or HTML documents. This action turns the sequence into a single XML document. See Basic usage (pg. 188) for how this works. This example is however not very useful. A much more common scenario, wrapping the results of a `p:for-each` loop, is shown in the Wrapping the results of a for-each loop (pg. 188) example.

Grouping

The `group-adjacent` option of `p:wrap-sequence` allows you to group documents together and wrap each group in the same wrapper element. The value of this option must a valid XPath expression. It is evaluated for every document in the input sequence. Documents with an equal result value are bundled together, resulting in a sequence of documents on the `result` port.

For all documents appearing on the `source` port:

- The document becomes the context item (accessible with the dot `.` operator).
- The expression in the `group-adjacent` option is evaluated.

  During this evaluation, the `position()` and `last()` functions are available to get the position of the document in the sequence and the size of the sequence.

- Two values computed by the XPath expression are considered equal if the XPath function `deep-equal()` (https://www.w3.org/TR/xpath-functions-31/#func-deep-equal) returns `true` for them. In most cases this simply means that values are equal when you intuitively expect them to be.
- All documents with equal values for the XPath expression, that are adjacent in the sequence, are wrapped in an element named by the `wrapper` option.

For an example see Grouping and wrapping (pg. 189).

Examples

Basic usage

The `source` port here receives a sequence of 4 `<fruit>` documents, by default. `p:wrap-sequence` wraps this into a `<fruits>` element, turning the four separate documents into a single one.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="carrot" color="orange"/>
    <fruit name="lemon" color="yellow"/>
  </p:input>
  <p:output port="result"/>

  <p:wrap-sequence wrapper="fruits"/>

</p:declare-step>
```

Result document:

```
<fruits>
   <fruit color="yellow" name="banana"/>
   <fruit color="orange" name="orange"/>
   <fruit color="orange" name="carrot"/>
   <fruit color="yellow" name="lemon"/>
</fruits>
```

Wrapping the results of a for-each loop

A very common scenario in which `p:wrap-sequence` is used, is in wrapping the results of a `p:for-each` loop. Such a loop usually results in a sequence of documents (one for each iteration). It's often easier to turn this (back) into a single document before continuing. The following example shows this. It has a `p:for-each` loop over all yellow fruit elements that adds an attribute `delivery="special"`. The resulting documents are wrapped in a `<yellow-fruits>` element, resulting in a single result document.

Source document:

```
<fruits>
   <fruit name="banana" color="yellow"/>
   <fruit name="orange" color="orange"/>
   <fruit name="carrot" color="orange"/>
   <fruit name="lemon" color="yellow"/>
</fruits>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true"/>
  <p:output port="result"/>

  <p:for-each>
    <p:with-input select="/*/*[@color eq 'yellow']"/>
    <p:add-attribute attribute-name="delivery" attribute-value="special"/>
  </p:for-each>

  <p:wrap-sequence wrapper="yellow-fruits"/>

</p:declare-step>
```

Result document:

```
<yellow-fruits>
   <fruit delivery="special" color="yellow" name="banana"/>
   <fruit delivery="special" color="yellow" name="lemon"/>
</yellow-fruits>
```

Grouping and wrapping

Like Basic usage (pg. 188), the `source` port here receives a sequence of 4 `<fruit>` documents. The first `p:wrap-sequence` step groups these, using the `color` attribute, and wraps these groups in a `<fruits-by-color>` element. This results in a sequence of 3 documents, which is wrapped again, to enable showing it as a single document.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">
    <fruit name="banana" color="yellow"/>
    <fruit name="orange" color="orange"/>
    <fruit name="carrot" color="orange"/>
    <fruit name="lemon" color="yellow"/>
  </p:input>
  <p:output port="result" sequence="true"/>

  <p:wrap-sequence wrapper="fruits-by-color" group-adjacent="/*/@color"/>
  <p:wrap-sequence wrapper="groups"/>

</p:declare-step>
```

Result document:

```
<groups>
   <fruits-by-color>
      <fruit color="yellow" name="banana"/>
   </fruits-by-color>
   <fruits-by-color>
      <fruit color="orange" name="orange"/>
      <fruit color="orange" name="carrot"/>
   </fruits-by-color>
   <fruits-by-color>
      <fruit color="yellow" name="lemon"/>
   </fruits-by-color>
</groups>
```

You might have expected that the result would group all fruits together by color, resulting in two documents: one for `banana`+`lemon` and one for `orange`+`carrot`. But `p:wrap-sequence` groups documents that are *adjacent* to each other only. It does *not* do what would be called "group by": all documents with the same value for the XPath expression together in a single group. If you need this, you will have to sort the documents first. But unfortunately, XProc does not have anything on board for that. For more complex grouping, the advice is to use XSLT or XQuery.

Additional details

- No document-properties of the source document(s) survive.
- The resulting document(s) have no `base-uri` property.

2.69 p:www-form-urldecode

Decode a URL parameter string into a map.

Summary

```
<p:declare-step type="p:www-form-urldecode">
  <output port="result" primary="true" content-types="application/json" sequence="true"/>
  <option name="value" as="xs:string" required="true"/>
</p:declare-step>
```

The `p:www-form-urldecode` step decodes a URL parameter string (like `a=b&c=d`) into a map. The result appears on the `result` port as a JSON document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`application/json`	`true`	The resulting map, as a JSON document.

Options:

Name	Type	Req?	Description
`value`	`xs:string`	`true`	The URL parameter string to decode.

Description

The `p:www-form-urldecode` step is one of the few steps that have no primary input port, its main input is the value of the `value` option. This value must be a valid URL parameter string: the part that usually comes after the `?` in a URL, like `a=b&c=d`. Officially, this is called a `x-www-form-urlencoded` string. This format is also used for sending HTML form data over HTTP.

The `p:www-form-urldecode` step takes such a string in its `value` option and converts it into a map. Each name/value pair in the input string is represented as a key/value pair in the map. Percent encoded values (like `%20` for space) are decoded. The `+` sign also acts as a space.

The resulting map appears on the `result` port as a JSON document.

There is also a step for *encoding* these kinds of strings, called `p:www-form-urlencode` (pg. 191).

Examples

Basic usage

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:www-form-urldecode value="a=b&amp;b=a%20b&amp;c=d+e+f"/>

</p:declare-step>
```

Result document:

```
{"a":"b","b":"a b","c":"d e f"}
```

Now assume you're interested in parameter `c` and want to turn its value into an XML element. This is how you can do this:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:www-form-urldecode value="a=b&amp;b=a%20b&amp;c=d+e+f"/>
  <p:identity>
    <p:with-input>
      <result-c>{.?c}</result-c>
    </p:with-input>
  </p:identity>

</p:declare-step>
```

Result document:

```
<result-c>d e f</result-c>
```

The document coming out of the `p:www-form-urldecode` step is a map. This map flows through my pipeline. The `p:identity` (pg. 78) step, since it directly follows `p:www-form-urldecode`, can address this map with the dot `.` operator. And `.?c` is syntactic sugar for: give me the value of the context item map with the key `'c'`. You could also have written it as `.('c')`.

Parameters with the same name

When the string to decode contains multiple parameters with the same name, the result for that key will be a *sequence* of values. We cannot serialize such a result directly (since JSON doesn't allow map entries with multiple values), but we can access and work with it in our program.

In the following example, parameter `a` is duplicated. The `p:identity` (pg. 78) step combines the values for `a` with a pipe character `|`, using the XPath `string-join() (https://www.w3.org/TR/xpath-functions-31/#func-string-join)` function.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:www-form-urldecode value="a=b&amp;b=a%20b&amp;a=d+e+f"/>
  <p:identity>
    <p:with-input>
      <result-a-sequence>{string-join(.?a, '|')}</result-a-sequence>
    </p:with-input>
  </p:identity>

</p:declare-step>
```

Result document:

```
<result-a-sequence>b|d e f</result-a-sequence>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- If any parameter name occurs more than once in the input string, a sequence will be associated with the respective key. The order in the sequence retains the order of name/value pairs in the encoded string. However, if this happens, you cannot serialize (write to, for instance, disk) the result because it is no longer valid JSON. You can however still use the map in your program and access its members. See also the Parameters with the same name (pg. 190) example

Errors raised

Error code	Description
`XC0037` (pg. 216)	It is a dynamic error if the `value` provided is not a properly `x-www-form-urlencoded` value.

2.70 p:www-form-urlencode

Encode parameters into a URL string.

Summary

```
<p:declare-step type="p:www-form-urlencode">
  <output port="result" primary="true" content-types="text/plain" sequence="true"/>
  <option name="parameters" as="map(xs:string,xs:anyAtomicType+)" required="true"/>
</p:declare-step>
```

The `p:www-form-urlencode` step encodes a set of parameters, given as entries in a map, into a URL parameter string (like `a=b&c=d`). The result appears on the `result` port as a text document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`result`	`output`	`true`	`text/plain`	`true`	The resulting URL parameter string, as a text document.

Options:

Name	Type	Req?	Description
`parameters`	`map(xs:string,xs:anyAtomicType+)`	`true`	A map with the parameters to encode.

Description

The `p:www-form-urlencode` step is one of the few steps that have no primary input port, its main input is the value of the `parameters` option. This value must be a map. The keys in the map are the parameter names, the value(s) the parameter values. The result will be a parameter string (the part that usually comes after the `?` in a URL, like `a=b&c=d`). Officially, this is called a `x-www-form-urlencoded` string. This format is also used for sending HTML form data over HTTP.

The resulting parameter string appears on the `result` port as a text document.

There is also a step for *decoding* these kinds of strings, called `p:www-form-urldecode` (pg. 189).

Examples

Basic usage

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:www-form-urlencode parameters="map{ 'a': 'b', 'c': 'd e f' }"/>

</p:declare-step>
```

Result document:

```
a=b&c=d+e+f
```

Multiple parameter values

If an entry in the map has multiple values (here for the entry with key `a`), the parameter is repeated in the output:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:output port="result"/>

  <p:www-form-urlencode parameters="map{ 'a': ('b', 'b2'), 'c': 'd e f' }"/>

</p:declare-step>
```

Result document:

```
a=b&a=b2&c=d+e+f
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).
- If an entry in the map has multiple values, an entry for each value will appear in the resulting URL string. See also the Multiple parameter values (pg. 192) example.

2.71 p:xinclude

Apply XInclude procesing to a document.

Summary

```
<p:declare-step type="p:xinclude">
  <input port="source" primary="true" content-types="xml html" sequence="true"/>
  <output port="result" primary="true" content-types="xml html" sequence="true"/>
  <option name="fixup-xml-base" as="xs:boolean" required="false" select="false()"/>
  <option name="fixup-xml-lang" as="xs:boolean" required="false" select="false()"/>
</p:declare-step>
```

The `p:xinclude` step applies XInclude (https://www.w3.org/TR/xinclude/) processing to the document appearing on the `source` document.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml` `html`	`true`	The document to apply the XInclude processing to.
`result`	`output`	`true`	`xml` `html`	`true`	The resulting document.

Options:

Name	Type	Req?	Default	Description
`fixup-xml-base`	`xs:boolean`	`false`	`false`	Perform base URI fixup (https://www.w3.org/TR/xinclude/#base) to the resulting document. Basically this means that `xml:base` elements are added to the root elements of the included files, containing the URIs they were included from. See also the Base URI fixup (pg. 194) example.
`fixup-xml-lang`	`xs:boolean`	`false`	`false`	Perform language fixup (https://www.w3.org/TR/xinclude/#language) on the resulting document. Basically this means that an `xml:lang` attribute is added to the root elements of the included documents. This stops the language settings of the including document from being inherited by the included document. See also the Language fixup (pg. 195) example.

Description

The XInclude (https://www.w3.org/TR/xinclude/) standard defines a syntax for specifying document inclusions. Basically this means:

- In your source document you write (zero, one or multiple times): `<xi:include href="…"/>` (the `xi` namespace prefix must be bound to the namespace `http://www.w3.org/2001/XInclude`).
- You run your document through the `p:xinclude` step.
- All `<xi:include>` elements are replaced with the contents of the document their `href` attribute is pointing to.
- All `<xi:include>` elements in the included documents are processed also, recursively.

Additionally, the XInclude (https://www.w3.org/TR/xinclude/) standard defines some additional attributes (https://www.w3.org/TR/xinclude/#include_element) for the `<xi:include>` element. The most used one is probably the `parse` attribute: `parse="xml"` (the default) means the document must be a well-formed XML document and is included as an XML fragment; `parse="text"` means the document is included as plain text. There is also an `<xi:fallback>` (https://www.w3.org/TR/xinclude/#fallback_element) child element that defines what will happen if the included document could not be found.

Examples

Basic usage

Remark upfront: some of the example documents contain `xml:lang` attributes. These are intended for the Language fixup (pg. 195) example below.

Assume we have a master document called `document-0.xml` that XIncludes two other documents:

```
<document-0 xmlns:xi="http://www.w3.org/2001/XInclude" xml:lang="en-us">
  <description>This is the master document</description>
  <xi:include href="includes/document-1.xml"/>
  <xi:include href="includes/document-2.xml"/>
</document-0>
```

The first include document `includes/document-1.xml` looks like this, please notice that it contains an `<xi:include>` element itself:

```
<document-1 xmlns:xi="http://www.w3.org/2001/XInclude">
  <description>This is include document 1</description>
  <xi:include href="document-2.xml"/>
</document-1>
```

The second include document `includes/document-2.xml` (that is also included by `includes/document-1.xml`) looks like this:

```
<document-1 xmlns:xi="http://www.w3.org/2001/XInclude">
  <description>This is include document 1</description>
  <xi:include href="document-2.xml"/>
</document-1>
```

Now if we run `document-0.xml` through the `p:xinclude` step, the result is as follows:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="document-0.xml"/>
  <p:output port="result"/>

  <p:xinclude/>

</p:declare-step>
```

Result document:

```
<document-0 xml:lang="en-us">
   <description>This is the master document</description>
   <document-1>
      <description>This is include document 1</description>
      <document-2 xml:lang="en-gb">
         <description>This is include document 2</description>
      </document-2>
   </document-1>
   <document-2 xml:lang="en-gb">
      <description>This is include document 2</description>
   </document-2>
</document-0>
```

Base URI fixup

The XInclude base URI fixup means that an `xml:base` attribute is added to the root elements of the included documents. Using the same documents and include structure as the Basic usage (pg. 194) example, it looks like this:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="document-0.xml"/>
  <p:output port="result"/>

  <p:xinclude fixup-xml-base="true"/>

</p:declare-step>
```

Result document:

```
<document-0 xml:lang="en-us">
   <description>This is the master document</description>
   <document-1 xml:base="file:/…/…/includes/document-1.xml">
      <description>This is include document 1</description>
      <document-2 xml:lang="en-gb" xml:base="file:/…/…/includes/document-2.xml">
         <description>This is include document 2</description>
      </document-2>
   </document-1>
   <document-2 xml:lang="en-gb" xml:base="file:/…/…/includes/document-2.xml">
      <description>This is include document 2</description>
   </document-2>
</document-0>
```

Notice that there is no `xml:base` attribute added to the root element of the master document. If you need *all* document root elements having an `xml:base` attribute, use the `p:add-xml-base` (pg. 8) step instead of XInclude base URI fixup:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="document-0.xml"/>
  <p:output port="result"/>

  <p:xinclude/>
  <p:add-xml-base relative="false"/>

</p:declare-step>
```

Result document:

```
<document-0 xml:lang="en-us" xml:base="file:/…/…/document-0.xml">
   <description>This is the master document</description>
   <document-1 xml:base="file:/…/…/includes/document-1.xml">
      <description>This is include document 1</description>
      <document-2 xml:lang="en-gb" xml:base="file:/…/…/includes/document-2.xml">
         <description>This is include document 2</description>
      </document-2>
   </document-1>
   <document-2 xml:lang="en-gb" xml:base="file:/…/…/includes/document-2.xml">
      <description>This is include document 2</description>
   </document-2>
</document-0>
```

Language fixup

The XInclude language fixup means that an `xml:lang` attribute is added to the root elements of the included documents. This stops the language settings of the including document from being inherited by the included documents. Using the same documents and include structure as the Basic usage (pg. 194) example, it looks like this:

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" href="document-0.xml"/>
  <p:output port="result"/>

  <p:xinclude fixup-xml-lang="true"/>

</p:declare-step>
```

Result document:

```
<document-0 xml:lang="en-us">
   <description>This is the master document</description>
   <document-1 xml:lang="">
      <description>This is include document 1</description>
      <document-2 xml:lang="en-gb">
         <description>This is include document 2</description>
      </document-2>
   </document-1>
   <document-2 xml:lang="en-gb">
      <description>This is include document 2</description>
   </document-2>
</document-0>
```

Notice the empty `xml:base=""` attribute on the `<document-1>` element. That wasn't there in the source. It stops the language settings of the including document `document-0.xml` (`xml:lang="en-us"`) from automatically being inherited by the included document `includes/document-1.xml`.

Additional details

- `p:xinclude` preserves all document-properties of the document(s) appearing on its `source` port.

Errors raised

Error code	Description
`XC0029` (pg. 216)	It is a dynamic error if an XInclude error occurs during processing.

2.72 p:xquery

Invoke an XQuery query.

Summary

```
<p:declare-step type="p:xquery">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <input port="query" primary="false" content-types="text xml" sequence="false"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="version" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:xquery` step applies an XQuery query to the sequence of documents provided on the `source` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The documents to invoke the XQuery query on, accessible using the XPath collection() (https://www.w3.org/TR/xpath-functions-31/#func-collection) function. The first document becomes the initial context item. If no documents are provided on the `source` port, the initial context item is undefined and the default collection is empty.
`result`	`output`	`true`	`any`	`true`	The resulting document(s).
`query`	`input`	`false`	`text xml`	`false`	The XQuery query this step invokes. There are several ways to specify the query, see "Specifying the query" on page 197 below.

Options:

Name	Type	Req?	Default	Description
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	A map with variable-names and corresponding values for the external variables in the query. See also the Passing parameters to a query (pg. 198) example.
`version`	`xs:string?`	`false`	`()`	Explicitly sets the XQuery version to use. Probable values are `3.0` or `3.1`. If this option is not set, officially the XQuery version used is implementation-defined and therefore depends on the XProc processor used. However, most likely the XProc processor will use the version as indicated in the query (the `xquery version` statement at the top of the query).

Description

XQuery is a programming language for querying sets of XML (and other) documents. More background information can be found on its Wikipedia page (https://en.wikipedia.org/wiki/XQuery). It is used in, for instance, XML databases like eXist (https://exist-db.org/exist/apps/homepage/index.html) and BaseX (https://basex.org/).

The `p:xquery` step invokes the XQuery query, as provided on the `query` port, on the document(s) appearing on the `source` port. The resulting document(s) appear on the `result` port.

Specifying the query

What appears on the `query` port determines the query the step invokes. XQuery queries are usually text documents (not XML), but there is a way to specify a query using XML-only called XQueryX (https://www.w3.org/TR/xqueryx-31/). There are several ways to specify the query:

- If the document on the `query` port is a text document, this is the query.
- If the document on the `query` port is an XML document with root element `<c:query>` (the `c` namespace prefix must be bound to the namespace `http://www.w3.org/ns/xproc-step`), the text value of this root element is the query.

  Usually this means that the `<c:query>` root element consists of text contents only (a single text node). However, if there are child elements, all text nodes in the document will be concatenated.
- If the document on the `query` port is an XML document with its root element in the XQueryX namespace (`http://www.w3.org/2005/XQueryX`), the document is treated as an XQueryX (https://www.w3.org/TR/xqueryx-31/) query.

  Whether XQueryX is supported is implementation defined and therefore depends on the XProc processor used.
- In all other cases, the document on the `query` port is first serialized (as if written to disk), using the (optional) `serialization` document-property settings. The resulting text is used as the query.

Examples

Basic usage

Assume we have input documents containing `<thing>` elements and we want to output a list of these elements, ordered by the value of their `id` attribute. The following XQuery query, called `sort-things.xql`, does the trick:

```
xquery version "3.0" encoding "UTF-8";

<things-sorted count="{count(collection()//thing)}">
  {
    for $thing in collection()//thing
      order by @id
    return
      $thing
  }
</things-sorted>
```

The following pipeline releases this query on some input documents containing `<thing>` elements:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">

    <things>
      <!-- Input document 1 -->
      <thing id="123"/>
      <nested-things>
        <thing id="456"/>
      </nested-things>
    </things>

    <things>
      <!-- Input document 2 -->
      <thing id="789"/>
    </things>

  </p:input>
  <p:output port="result" sequence="true"/>

  <p:xquery>
    <p:with-input port="query" href="sort-things.xql"/>
  </p:xquery>

</p:declare-step>
```

Result document:

```
<things-sorted count="3">
   <thing id="123"/>
   <thing id="456"/>
   <thing id="789"/>
</things-sorted>
```

Passing parameters to a query

Let's make the Basic usage (pg. 197) example also usable for other elements than `<thing>`. The name of the element to count is passed as an external variable. The query, called `sort.xql`, is as follows:

```
xquery version "3.0" encoding "UTF-8";

declare variable $elm-name as xs:string external;

<things-sorted count="{count(collection()//*[local-name() eq $elm-name])}">
  {
    for $elm in collection()//*[local-name() eq $elm-name]
      order by @id
    return
      $elm
  }
</things-sorted>
```

The following pipeline releases this query on the same input documents as the Basic usage (pg. 197) example. It again selects `<thing>` elements, but the (local) name of this element is now passed as an external variable:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source" sequence="true">

    <things>
      <!-- Input document 1 -->
      <thing id="123"/>
      <nested-things>
        <thing id="456"/>
      </nested-things>
    </things>

    <things>
      <!-- Input document 2 -->
      <thing id="789"/>
    </things>

  </p:input>
  <p:output port="result" sequence="true"/>

  <p:xquery parameters="map{'elm-name': 'thing'}">
    <p:with-input port="query" href="sort.xql"/>
  </p:xquery>

</p:declare-step>
```

Result document:

```
<things-sorted count="3">
   <thing id="123"/>
   <thing id="456"/>
   <thing id="789"/>
</things-sorted>
```

Additional details

- Which XQuery version(s) is/are supported is implementation-defined and therefore depends on the XProc processor used.
- No document-properties from the source document(s) are preserved.
- The `base-uri` document of each result document is determined by the query. If the query does not establish a base URI, the document will not have a `base-uri` document-property.

Errors raised

Error code	Description
`XC0009` (pg. 216)	It is a dynamic error if the specified XQuery version is not available.
`XC0101` (pg. 217)	It is a dynamic error if a document appearing on port `source` cannot be represented in the XDM version associated with the chosen XQuery version, e.g. when a JSON document contains a map and XDM 3.0 is used.
`XC0102` (pg. 217)	It is a dynamic error if any key in option `parameters` is associated to a value that cannot be represented in the XDM version associated with the chosen XQuery version, e.g. with a map, an array, or a function when XDM 3.0 is used.

Error code	Description
XC0103 (pg. 217)	It is a dynamic error if any error occurs during XQuery's static analysis phase.
XC0104 (pg. 217)	It is a dynamic error if any error occurs during XQuery's dynamic evaluation phase.

2.73 p:xsl-formatter

Renders an XSL-FO document.

Summary

```
<p:declare-step type="p:xsl-formatter">
  <input port="source" primary="true" content-types="xml" sequence="false"/>
  <output port="result" primary="true" content-types="any" sequence="false"/>
  <option name="content-type" as="xs:string?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
</p:declare-step>
```

The `p:xsl-formatter` step expects a valid XSL-FO (https://en.wikipedia.org/wiki/XSL_Formatting_Objects) document on its `source` port. This is rendered, usually into PDF. The resulting rendition appears, as a binary document, on the `result` port.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`xml`	`false`	The XSL-FO (https://en.wikipedia.org/wiki/XSL_Formatting_Objects) document to render.
`result`	`output`	`true`	`any`	`false`	The resulting rendition.

Options:

Name	Type	Req?	Default	Description
`content-type`	`xs:string?`	`false`	`()`	The content-type (media type) of the rendition that appears on the `result` port. The default value is `application/pdf`. Whether any other formats are supported is implementation-defined and therefore dependent on the XProc processor and renderer used.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	Parameters used to control the rendering. The XProc specification does not define any parameters for this option. A specific XProc processor (or renderer used) might define its own.

Description

The `p:xsl-formatter` step allows you to transform XML into some kind of rendition, usually PDF. To do this, you must first transform your XML into XSL-FO (https://en.wikipedia.org/wiki/XSL_Formatting_Objects). This can be done by several means, most likely one or more XSLT transformations by `p:xslt` (pg. 200). After this, the `p:xsl-formatter` step renders the document for you.

In most cases, `p:xsl-formatter` relies on an external XSL-FO (https://en.wikipedia.org/wiki/XSL_Formatting_Objects) formatter, for instance the open source FOP (https://xmlgraphics.apache.org/fop/) or one of the commercial ones. You'll probably have to do some XProc processor dependent configuration before this step will work. Please consult the XProc processor documentation about this.

Examples

Basic usage

The following pipeline transforms an XSL-FO document into PDF using `p:xsl-formatter`, and stores it as `result.pdf`:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="result-uri@store-pdf"/>

  <p:xsl-formatter/>
  <p:store href="result.pdf" name="store-pdf"/>

</p:declare-step>
```

Additional details

- The document appearing on the `result` port only has a `content-type` property. It has no other document-properties (also no `base-uri`).

Errors raised

Error code	Description
`XC0167` (pg. 219)	It is a dynamic error if the requested document cannot be produced.
`XC0204` (pg. 220)	It is a dynamic error if the requested content-type is not supported.
`XD0079` (pg. 220)	It is a dynamic error if a supplied content-type is not a valid media type of the form " *`type/subtype+ext`* " or " *`type/subtype`* ".

2.74 p:xslt

Invoke an XSLT stylesheet.

Summary

```
<p:declare-step type="p:xslt">
  <input port="source" primary="true" content-types="any" sequence="true"/>
  <output port="result" primary="true" content-types="any" sequence="true"/>
  <input port="stylesheet" primary="false" content-types="xml" sequence="false"/>
  <output port="secondary" primary="false" content-types="any" sequence="true"/>
  <option name="global-context-item" as="item()?" required="false" select="()"/>
  <option name="initial-mode" as="xs:QName?" required="false" select="()"/>
  <option name="output-base-uri" as="xs:anyURI?" required="false" select="()"/>
  <option name="parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="populate-default-collection" as="xs:boolean?" required="false" select="true()"/>
  <option name="static-parameters" as="map(xs:QName, item()*)?" required="false" select="()"/>
  <option name="template-name" as="xs:QName?" required="false" select="()"/>
  <option name="version" as="xs:string?" required="false" select="()"/>
</p:declare-step>
```

The `p:xslt` step invokes the XSLT stylesheet that appears on the `stylesheet` port. What exactly happens depends on the XSLT version used.

Ports:

Port	Type	Primary?	Content types	Seq?	Description
`source`	`input`	`true`	`any`	`true`	The source document(s) to transform. What exactly happens with these documents depends on the XSLT stylesheet version. See below.
`result`	`output`	`true`	`any`	`true`	The principal resulting document(s) of the transformation.
`stylesheet`	`input`	`false`	`xml`	`false`	The XSLT stylesheet to invoke.
`secondary`	`output`	`false`	`any`	`true`	Any secondary documents created by the transformation. Starting with XSLT version 2.0, you can use the XSLT `<xsl:result-document>` instruction for this.

Options:

Name	Type	Req?	Default	Description
`global-context-item`	`item()?`	`false`	`()`	This explicitly sets the global context item for the XSLT stylesheet: the data the stylesheet starts working on. If you don't use this option, the global context item is determined by what appears on the `source` port. Setting the global context item is supported starting XSLT version 3.0.
`initial-mode`	`xs:QName?`	`false`	`()`	If this option is set, the XSLT stylesheet starts processing in the given mode. Modes are supported starting XSLT version 2.0.
`output-base-uri`	`xs:anyURI?`	`false`	`()`	Explicitly sets the base URI for the stylesheet result(s). What exactly happens depends on the XSLT stylesheet version. See below.
`parameters`	`map(xs:QName, item()*)?`	`false`	`()`	A map with parameter-names and corresponding values to pass as global parameters to the XSLT stylesheet.
`populate-default-collection`	`xs:boolean?`	`false`	`true`	XSLT stylesheets have a *default collection*, accessible using the XPath collection() (https://www.w3.org/TR/xpath-functions-31/#func-collection) function. If you set this option to `true`, the documents appearing on the `source` port become the default collection. Collections are supported starting XSLT version 2.0.
`static-parameters`	`map(xs:QName, item()*)?`	`false`	`()`	A map with parameter-names and corresponding values to pass as static parameters to the XSLT stylesheet. Static stylesheet parameters are supported starting XSLT version 3.0.
`template-name`	`xs:QName?`	`false`	`()`	Usually, an XSLT stylesheet starts processing using "apply-template invocation": it tries to find the most appropriate matching template and starts processing there. However, if the `template-name` option is set, a "call-template invocation" is performed: processing starts at that named template. Starting processing at a named template is supported starting XSLT version 2.0.
`version`	`xs:string?`	`false`	`()`	Explicitly sets the XSLT stylesheet version. Probable values are `1.0`, `2.0` or `3.0`. If this option is not set, officially the XSLT version used is implementation-defined and therefore depends on the XProc processor used. However, most likely the XProc processor will use the stylesheet version as indicated on the stylesheet root element (the `xsl:stylesheet/@version` or `xsl:transform/@version` attribute).

Description

The `p:xslt` step invokes the XSLT stylesheet that appears on the `stylesheet` port. What is used as input, how the XSLT processing starts and where/how the results appear depends on the XSLT version used. This is explained in the sections below.

Because of all the details, invoking the `p:xslt` step seems complicated. However, presumably, in the vast majority of cases it will be used in a classical manner: invoke an XSLT stylesheet on a source document and continue the pipeline using its result. Maybe with some parameters, maybe with some secondary results. For this, have a look at the Basic usage (pg. 204) and Basic usage with secondary documents (pg. 205) examples and don't let all the details overwhelm you.

Specifying the XSLT version is important but, in most cases, rather simple: most likely the version as specified on the XSLT stylesheet root element (the `xsl:stylesheet/@version` or `xsl:transform/@version` attribute) is used. Since such a version attribute is required anyway, there usually won't be anything special you need to do. However, if you want you can set the version explicitly using the `version` option.

Invoking an XSLT 3.0 stylesheet

If the stylesheet version is determined as 3.0, the following happens:

- The parameters as set by the `static-parameters` option are passed to the stylesheet invocation as values for its static parameters.
- An XSLT version 3.0 stylesheet has a *global context item*, the data the stylesheet works upon. This is determined as follows:
  - If the `global-context-item` option is set, this becomes the global context item.
  - If the `global-context-item` option is *not* set *and* a single document appears on the `source` port, this will become the global context item.
  - If the `global-context-item` option is *not* set *and* none or multiple documents appear on the `source` port, the global context item is absent/empty.
- If the `populate-default-collection` is set to `true`, all documents that appear on the `source` port become the *default collection*, accessible using the XPath collection() (https://www.w3.org/TR/xpath-functions-31/#func-collection) function.
- Then it is determined how to start the stylesheet processing:
  - If the `template-name` is *not* set, the normal "apply-template invocation" is performed. The document(s) that appear on the `source` port are used, one by one, for the initial match.

    If the `initial-mode` option is set, processing starts in that mode.
  - If the `template-name` is set, the named template with that name (`<xsl:template name="…">`) is invoked.

    The `initial-mode` option is ignored.
- The stylesheet processes.
- The result(s) appears on the output port(s):
  - All principal results of the stylesheet appear on the `result` port.
  - Any results created by `<xsl:result-document>` instructions appear on the `secondary` port.
- Finally, the base URIs of the resulting documents (their `base-uri` document-property values) are determined. For this we first need to determine the *base-output-URI*:
  - If the `base-output-uri` option is set, this value is used as base-output-URI.
  - If the `base-output-uri` option is *not* set *and* there are documents on the `source` port, the base URI of the *first* document on the `source` port is used as base output URI.
  - If the `base-output-uri` option is *not* set *and* there are *no* documents on the `source` port, the base URI of the stylesheet is used as base-output-URI.

  The base URIs of the resulting documents (their `base-uri` document-property values) are now computed using this base-output-URI:
  - The base URI of the principal output document(s) becomes the base-output-URI. This means that when there are multiple principal documents, they all have the same base URI!
  - For all documents appearing on the `secondary` port, the base URI is determined by the `xsl:result-document/@href` attribute. A relative value is made absolute against the base-output-URI.

Invoking an XSLT 2.0 stylesheet

If the stylesheet version is determined as 3.0, the following happens:

- The following options are ignored: `static-parameters`, `global-context-item`.
- An XSLT version 2.0 stylesheet has an *initial context node*, the initial data the stylesheet works upon. This is determined as follows:
  - When no documents appear on the `source` port, the initial context node is undefined/empty.
  - When one or multiple documents appear on the `source` port, only the *first* document becomes the initial context node.

- If the `populate-default-collection` is set to `true`, all documents that appear on the `source` port become the *default collection*, accessible using the XPath collection() (https://www.w3.org/TR/xpath-functions-31/#func-collection) function.
- Then it is determined how to start the stylesheet processing:
  - If the `template-name` is *not* set, the normal "apply-template invocation" is performed. The document(s) that appear on the `source` port are used, one by one, for the initial match.

    If the `initial-mode` option is set, processing starts in that mode.
  - If the `template-name` is set, the named template with that name (`<xsl:template name="…">`) is invoked.

    The `initial-mode` option is ignored.
- The stylesheet processes.
- The result(s) appears on the output port(s):
  - The principal result document of the stylesheet appears on the `result` port.
  - Any results created by `<xsl:result-document>` instructions appear on the `secondary` port.
- Finally, the base URIs of the resulting documents (their `base-uri` document-property values) are determined. For this we first need to determine the *base-output-URI*:
  - If the `base-output-uri` option is set, this value is used as base-output-URI.
  - If the `base-output-uri` option is *not* set *and* there are documents on the `source` port, the base URI of the *first* document on the `source` port is used as base output URI.
  - If the `base-output-uri` option is *not* set *and* there are *no* documents on the `source` port, the base URI of the stylesheet is used as base-output-URI.

  The base URIs of the resulting documents (their `base-uri` document-property values) are now computed using this base-output-URI:
  - The base URI of the principal output document becomes the base-output-URI.
  - For all documents appearing on the `secondary` port, the base URI is determined by the `xsl:result-document/@href` attribute. A relative value is made absolute against the base-output-URI.

Invoking an XSLT 1.0 stylesheet

If the stylesheet version is determined as 1.0, the following happens:

- The following options are ignored: `global-context-item`, `initial-mode`, `populate-default-collection`, `static-parameters`, `template-name`.
- There must be exactly one document appearing on the `source` port. This document will be processed.
- The stylesheet processes.
- The resulting document appears on the `result` port. The `secondary` port will always be empty.
- Finally, the base URI of the resulting document (its `base-uri` document-property value) is determined. For this we first need to determine the *base-output-URI*:
  - If the `base-output-uri` option is set, this value is used as base-output-URI.
  - If the `base-output-uri` option is *not* set, the base URI of the *first* document on the `source` port is used as base-output-URI.

  The base URI of the output document becomes the base-output-URI.

Examples

Basic usage

For the following example, we'll use a very simple (3.0) stylesheet, called `add-comment.xsl`, that adds a comment as the first child of the root element:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xs="http://
www.w3.org/2001/XMLSchema" expand-text="true">

  <xsl:mode on-no-match="shallow-copy"/>

  <xsl:param name="comment-text" as="xs:string" required="false" select="'This is an added comment'"/>

  <xsl:template match="/*">
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
      <xsl:comment> == {current-dateTime()} - {$comment-text} == </xsl:comment>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Running this without any bells and whistles is as follows:

Source document:

```
<customers>
   <customer>
      <name>PXSLT Company Ltd</name>
   </customer>
</customers>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:xslt>
    <p:with-input port="stylesheet" href="add-comment.xsl"/>
  </p:xslt>

</p:declare-step>
```

Result document:

```
<customers><!-- == 2025-06-25T13:24:56.1797688+02:00 - This is an added comment == -->
   <customer>
      <name>PXSLT Company Ltd</name>
   </customer>
</customers>
```

Setting a stylesheet parameter is done by supplying a map with parameter name/value pairs as the value of the `parameters` option:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:xslt parameters="map{'comment-text': 'Special comment text by parameter!'}">
    <p:with-input port="stylesheet" href="add-comment.xsl"/>
  </p:xslt>

</p:declare-step>
```

Result document:

```
<customers><!-- == 2025-06-25T13:24:56.2053384+02:00 - Special comment text by parameter! == -->
   <customer>
      <name>PXSLT Company Ltd</name>
   </customer>
</customers>
```

Basic usage with secondary documents

The output of `<xsl:result-document>` stylesheet instructions is written to the `secondary` port of the `p:xslt` invocation. The base URI of these documents is the value of the `xsl:result-document/@href` attribute.

The following stylesheet, called `split-documents.xsl`, writes the contents of each `<document>` element to a separate, secondary, document. The base URI of the output documents is inferred from the `document/@name` attribute. The primary output of the stylesheet is almost identical to its input: the full URI of each written secondary output document is added to the `<document>` element in an `href` attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xs="http://
www.w3.org/2001/XMLSchema" expand-text="true">

  <xsl:mode on-no-match="shallow-copy"/>

  <xsl:template match="document">
    <xsl:variable name="href" as="xs:string" select="resolve-uri('tmp/' || @name)"/>
    <xsl:result-document href="{$href}">
      <xsl:sequence select="*"/>
    </xsl:result-document>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:attribute name="href" select="$href"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

When using this stylesheet in an XProc pipeline, the documents, written by `<xsl:result-document>` instructions, end up on the `secondary` port. If we want these written to disk, we need to add some code for it.

Source document:

```
<documents>
   <document name="x1.xml">
      <document-1/>
   </document>
   <document name="x2.xml">
      <document-2/>
   </document>
</documents>
```

Pipeline document:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0">

  <p:input port="source"/>
  <p:output port="result" pipe="result@create-secondary-documents"/>

  <p:xslt name="create-secondary-documents">
    <p:with-input port="stylesheet" href="split-documents.xsl"/>
  </p:xslt>

  <p:for-each>
    <p:with-input pipe="secondary"/>
    <p:store href="{base-uri(/)}"/>
  </p:for-each>

</p:declare-step>
```

Result document:

```
<documents>
   <document name="x1.xml" href="file:/…/…/tmp/x1.xml">
      <document-1/>
   </document>
   <document name="x2.xml" href="file:/…/…/tmp/x2.xml">
      <document-2/>
   </document>
</documents>
```

The primary output of the `p:xslt` is explicitly piped to the output port of the pipeline here (by the `p:output/@pipe` attribute).

The `p:for-each` after the `p:xslt` iterates over all secondary documents and invokes `p:store` (pg. 132) to store them to disk, using their base URI, that was set by the stylesheet. The result will be two documents, called `x1.xml` and `x2.xml`, in the `tmp/` folder underneath the stylesheet location.

Additional details

- Which XSLT version(s) is/are supported is implementation-defined and therefore depends on the XProc processor used. In most cases at least version 3.0 will be supported.
- No document-properties from the source document(s) are preserved.
- The `base-uri` document of each result document (both for the `result` and the `secondary` port) is determined by the transformation. If the transformation does not establish a base URI, the document will not have a `base-uri` document-property.
- If the `template-name` option is set, the `initial-mode` option is ignored.
- A relative value for the `output-base-uri` option is made absolute against the base URI of the element in the pipeline it is specified on. In most cases this will be the path of the pipeline document.
- An XSLT stylesheet can terminate processing using an `<xsl:message terminate="true">` instruction. How such a termination is reported by the XProc processor is implementation-defined and therefore depends on the XProc processor used.
- The order in which result documents appear on the `secondary` port is implementation-defined and therefore depends on the XProc processor used.

Errors raised

Error code	Description
`XC0007` (pg. 216)	It is a dynamic error if any key in `parameters` is associated to a value which is not an instance of the XQuery 1.0 and XPath 2.0 Data Model, e.g. with a map, an array, or a function.
`XC0008` (pg. 216)	It is a dynamic error if the stylesheet does not support a given mode.
`XC0038` (pg. 216)	It is a dynamic error if the specified xslt version is not available.
`XC0039` (pg. 216)	It is a dynamic error if the source port does not contain exactly one XML document or one HTML document if XSLT 1.0 is used.
`XC0056` (pg. 216)	It is a dynamic error if the stylesheet does not provide a given template.
`XC0093` (pg. 217)	It is a dynamic error if a static error occurs during the static analysis of the XSLT stylesheet.
`XC0094` (pg. 217)	It is a dynamic error if any document supplied on the source port is not an XML document, an HTML documents, or a Text document if XSLT 2.0 is used.
`XC0095` (pg. 217)	It is a dynamic error if an error occurred during the transformation.
`XC0096` (pg. 217)	It is a dynamic error if the transformation is terminated by XSLT message termination.
`XC0105` (pg. 218)	It is a dynamic error if an XSLT 1.0 stylesheet is invoked and option `parameters` contains a value that is not an atomic value or a node.
`XC0121` (pg. 218)	It is a dynamic error if a document appearing on the `secondary` port has a base URI that is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .

3 Categories

3.1 Overview

Primary categories

The following categories are defined by the XProc specification itself:

- Standard XProc steps (pg. 208)
  XProc steps that a conformant XProc processor *must* support. In other words: whatever processor you use for processing your XProc pipelines, you can trust that these steps will always work.
- XProc dynamic pipeline execution steps (pg. 210)
  XProc steps that deal with dynamic execution of pipelines (running pipelines inside another pipeline).
- XProc email related steps (pg. 210)
  XProc steps that deal with email. An XProc processor is not required to support these.
- XProc file and directory related steps (pg. 210)
  XProc steps that deal with files and directories. An XProc processor is not required to support these.
- XProc Invisible XML related steps (pg. 210)
  XProc steps that deal with the processing of Invisible XML. An XProc processor is not required to support these.
- XProc operating system related steps (pg. 210)
  XProc steps that deal with the operating system. An XProc processor is not required to support these.
- XProc paged media related steps (pg. 210)
  XProc steps that deal with creating paged media, for instance PDF. An XProc processor is not required to support these.
- XProc text related steps (pg. 211)
  XProc steps that deal with text, for instance Markdown. An XProc processor is not required to support these.
- XProc validation related steps (pg. 211)
  XProc steps that deal with validation of XML and JSON documents. An XProc processor is not required to support these.

Other categories

The following categories are defined by XProcRef:

- Additional standards (pg. 211)
  Steps that implement additional standards.
- Archive handling (pg. 212)
  These steps handle archives (for instance, ZIP files).
- Base URI related (pg. 212)
  These steps act on or use the base URI value of elements.
- Basic XML manipulation (pg. 212)
  These steps implement basic XML manipulation, like adding attributes, inserting elements, etc.
- Compression (pg. 213)
  Steps that have to do with compressing (or uncompressing) documents.
- Interaction with the environment (pg. 213)
  Steps that interact (or prepare interaction) with the environment of the pipeline, for instance communication over HTTP(S) or working with directories and files.
- JSON related steps (pg. 214)
  Steps for handling JSON documents.
- Miscellaneous (pg. 214)
  Miscellaneous steps, which are usually used for housekeeping purposes in the pipeline.
- Namespace handling (pg. 215)

Steps that have to do with namespace handling.

- Text document related steps (pg. 215)
  Steps that do something with (lines in) text documents.

3.2 Standard XProc steps

XProc steps that a conformant XProc processor *must* support. In other words: whatever processor you use for processing your XProc pipelines, you can trust that these steps will always work.

A

- `p:add-attribute` (pg. 6) - Add (or replace) an attribute on a set of elements.
- `p:add-xml-base` (pg. 8) - Add explicit xml:base attributes to a document.
- `p:archive` (pg. 11) - Perform operations on archive files.
- `p:archive-manifest` (pg. 18) - Create an XML manifest document describing the contents of an archive file.

C

- `p:cast-content-type` (pg. 22) - Changes the media type of a document.
- `p:compare` (pg. 29) - Compares documents for equality.
- `p:compress` (pg. 31) - Compresses a document.
- `p:count` (pg. 32) - Count the number of documents.

D

- `p:delete` (pg. 36) - Delete nodes in documents.

E

- `p:encode` (pg. 45) - Encodes a document.
- `p:error` (pg. 47) - Raises an error.

F

- `p:filter` (pg. 63) - Selects parts of a document.

H

- `p:hash` (pg. 65) - Computes a hash code for a value.
- `p:http-request` (pg. 68) - Interact using HTTP (or related protocols).

I

- `p:identity` (pg. 78) - Copies the source to the result without modifications.
- `p:insert` (pg. 80) - Inserts one document into another.

J

- `p:json-join` (pg. 86) - Joins documents into a JSON array document.
- `p:json-merge` (pg. 89) - Joins documents into a JSON map document.

L

- `p:label-elements` (pg. 92) - Labels elements by adding an attribute.
- `p:load` (pg. 94) - Loads a document.

M

N

P

R

S

T

U

W

X

- `p:xinclude` (pg. 193) - Apply XInclude procesing to a document.
- `p:xquery` (pg. 196) - Invoke an XQuery query.
- `p:xslt` (pg. 200) - Invoke an XSLT stylesheet.

3.3 XProc dynamic pipeline execution steps

XProc steps that deal with dynamic execution of pipelines (running pipelines inside another pipeline).

- `p:run` (pg. 116) - Runs a dynamically loaded pipeline.

3.4 XProc email related steps

XProc steps that deal with email. An XProc processor is not required to support these.

- `p:send-mail` (pg. 121) - Sends an email message.

3.5 XProc file and directory related steps

XProc steps that deal with files and directories. An XProc processor is not required to support these.

D

- `p:directory-list` (pg. 37) - List the contents of a directory.

F

- `p:file-copy` (pg. 49) - Copies a file or directory.
- `p:file-create-tempfile` (pg. 51) - Creates a temporary file.
- `p:file-delete` (pg. 54) - Deletes a file or directory.
- `p:file-info` (pg. 56) - Returns information about a file or directory.
- `p:file-mkdir` (pg. 59) - Creates a directory.
- `p:file-move` (pg. 60) - Moves or renames a file or directory.
- `p:file-touch` (pg. 62) - Changes the modification timestamp of a file.

3.6 XProc Invisible XML related steps

XProc steps that deal with the processing of Invisible XML. An XProc processor is not required to support these.

- `p:invisible-xml` (pg. 83) - Performs invisible XML processing.

3.7 XProc operating system related steps

XProc steps that deal with the operating system. An XProc processor is not required to support these.

- `p:os-exec` (pg. 106) - Runs an external command.
- `p:os-info` (pg. 110) - Returns information about the operating system.

3.8 XProc paged media related steps

XProc steps that deal with creating paged media, for instance PDF. An XProc processor is not required to support these.

C

- `p:css-formatter` (pg. 34) - Renders a document using CSS formating.

X

- `p:xsl-formatter` (pg. 199) - Renders an XSL-FO document.

3.9 XProc text related steps

XProc steps that deal with text, for instance Markdown. An XProc processor is not required to support these.

- `p:markdown-to-html` (pg. 99) - Converts a Markdown document into HTML.

3.10 XProc validation related steps

XProc steps that deal with validation of XML and JSON documents. An XProc processor is not required to support these.

- `p:validate-with-dtd` (pg. 158) - Validates a document using a DTD.
- `p:validate-with-json-schema` (pg. 160) - Validates a JSON document using JSON schema.
- `p:validate-with-nvdl` (pg. 164) - Validate a document using NVDL.
- `p:validate-with-relax-ng` (pg. 167) - Validate a document using RELAX NG.
- `p:validate-with-schematron` (pg. 171) - Validates a document using Schematron.
- `p:validate-with-xml-schema` (pg. 176) - Validates a document using XML Schema.

3.11 Additional standards

Steps that implement additional standards.

C

- `p:css-formatter` (pg. 34) - Renders a document using CSS formating.

H

- `p:http-request` (pg. 68) - Interact using HTTP (or related protocols).

I

- `p:invisible-xml` (pg. 83) - Performs invisible XML processing.

M

- `p:markdown-to-html` (pg. 99) - Converts a Markdown document into HTML.

V

- `p:validate-with-dtd` (pg. 158) - Validates a document using a DTD.
- `p:validate-with-json-schema` (pg. 160) - Validates a JSON document using JSON schema.
- `p:validate-with-nvdl` (pg. 164) - Validate a document using NVDL.
- `p:validate-with-relax-ng` (pg. 167) - Validate a document using RELAX NG.
- `p:validate-with-schematron` (pg. 171) - Validates a document using Schematron.
- `p:validate-with-xml-schema` (pg. 176) - Validates a document using XML Schema.

X

- `p:xinclude` (pg. 193) - Apply XInclude procesing to a document.
- `p:xquery` (pg. 196) - Invoke an XQuery query.
- `p:xsl-formatter` (pg. 199) - Renders an XSL-FO document.
- `p:xslt` (pg. 200) - Invoke an XSLT stylesheet.

3.12 Archive handling

These steps handle archives (for instance, ZIP files).

A

- `p:archive` (pg. 11) - Perform operations on archive files.
- `p:archive-manifest` (pg. 18) - Create an XML manifest document describing the contents of an archive file.

U

- `p:unarchive` (pg. 147) - Extracts documents from an archive file.

3.13 Base URI related

These steps act on or use the base URI value of elements.

A

- `p:add-attribute` (pg. 6) - Add (or replace) an attribute on a set of elements.
- `p:add-xml-base` (pg. 8) - Add explicit xml:base attributes to a document.

S

- `p:set-attributes` (pg. 124) - Add (or replace) attributes on a set of elements.
- `p:set-properties` (pg. 126) - Sets or changes document-properties.

3.14 Basic XML manipulation

These steps implement basic XML manipulation, like adding attributes, inserting elements, etc.

A

- `p:add-attribute` (pg. 6) - Add (or replace) an attribute on a set of elements.

D

- `p:delete` (pg. 36) - Delete nodes in documents.

I

- `p:insert` (pg. 80) - Inserts one document into another.

L

- `p:label-elements` (pg. 92) - Labels elements by adding an attribute.

N

- `p:namespace-delete` (pg. 101) - Deletes namespaces from a document.
- `p:namespace-rename` (pg. 103) - Renames a namespace to a new URI.

R

- `p:rename` (pg. 113) - Renames nodes in a document.
- `p:replace` (pg. 115) - Replace nodes with a document.

S

- `p:set-attributes` (pg. 124) - Add (or replace) attributes on a set of elements.
- `p:string-replace` (pg. 134) - Replaces nodes with strings.

U

- `p:unwrap` (pg. 154) - Unwraps elements in a document.
- `p:uuid` (pg. 156) - Injects UUIDs into a document.

W

- `p:wrap` (pg. 184) - Wraps nodes in a parent element.

3.15 Compression

Steps that have to do with compressing (or uncompressing) documents.

A

- `p:archive` (pg. 11) - Perform operations on archive files.
- `p:archive-manifest` (pg. 18) - Create an XML manifest document describing the contents of an archive file.

C

- `p:compress` (pg. 31) - Compresses a document.

U

- `p:unarchive` (pg. 147) - Extracts documents from an archive file.
- `p:uncompress` (pg. 152) - Uncompresses a document.

3.16 Interaction with the environment

Steps that interact (or prepare interaction) with the environment of the pipeline, for instance communication over HTTP(S) or working with directories and files.

D

- `p:directory-list` (pg. 37) - List the contents of a directory.

F

- `p:file-copy` (pg. 49) - Copies a file or directory.
- `p:file-create-tempfile` (pg. 51) - Creates a temporary file.
- `p:file-delete` (pg. 54) - Deletes a file or directory.
- `p:file-info` (pg. 56) - Returns information about a file or directory.
- `p:file-mkdir` (pg. 59) - Creates a directory.
- `p:file-move` (pg. 60) - Moves or renames a file or directory.
- `p:file-touch` (pg. 62) - Changes the modification timestamp of a file.

H

- `p:http-request` (pg. 68) - Interact using HTTP (or related protocols).

L

- `p:load` (pg. 94) - Loads a document.

M

- `p:make-absolute-uris` (pg. 97) - Make URIs in the document absolute.

O

- `p:os-exec` (pg. 106) - Runs an external command.
- `p:os-info` (pg. 110) - Returns information about the operating system.

S

- `p:send-mail` (pg. 121) - Sends an email message.
- `p:store` (pg. 132) - Stores a document.

W

- `p:www-form-urldecode` (pg. 189) - Decode a URL parameter string into a map.
- `p:www-form-urlencode` (pg. 191) - Encode parameters into a URL string.

3.17 JSON related steps

Steps for handling JSON documents.

J

- `p:json-join` (pg. 86) - Joins documents into a JSON array document.
- `p:json-merge` (pg. 89) - Joins documents into a JSON map document.

V

- `p:validate-with-json-schema` (pg. 160) - Validates a JSON document using JSON schema.

3.18 Miscellaneous

Miscellaneous steps, which are usually used for housekeeping purposes in the pipeline.

C

- `p:cast-content-type` (pg. 22) - Changes the media type of a document.
- `p:compare` (pg. 29) - Compares documents for equality.
- `p:count` (pg. 32) - Count the number of documents.

E

- `p:encode` (pg. 45) - Encodes a document.
- `p:error` (pg. 47) - Raises an error.

F

- `p:filter` (pg. 63) - Selects parts of a document.

H

- `p:hash` (pg. 65) - Computes a hash code for a value.

I

- `p:identity` (pg. 78) - Copies the source to the result without modifications.

M

- `p:make-absolute-uris` (pg. 97) - Make URIs in the document absolute.
- `p:message` (pg. 100) - Produces a message.

P

- `p:pack` (pg. 111) - Merges two document sequences, pair-wise.

S

- `p:set-properties` (pg. 126) - Sets or changes document-properties.
- `p:sink` (pg. 127) - Discards all source documents.
- `p:sleep` (pg. 127) - Delays the execution of the pipeline.
- `p:split-sequence` (pg. 128) - Splits a sequence of documents.

U

- `p:uuid` (pg. 156) - Injects UUIDs into a document.

W

- `p:wrap-sequence` (pg. 187) - Wraps a sequence of documents in an element.

3.19 Namespace handling

Steps that have to do with namespace handling.

- `p:namespace-delete` (pg. 101) - Deletes namespaces from a document.
- `p:namespace-rename` (pg. 103) - Renames a namespace to a new URI.

3.20 Text document related steps

Steps that do something with (lines in) text documents.

- `p:text-count` (pg. 138) - Counts the number of lines in a text document.
- `p:text-head` (pg. 139) - Returns lines from the beginning of a text document.
- `p:text-join` (pg. 140) - Concatenates text documents.
- `p:text-replace` (pg. 142) - Replace substrings in a text document.
- `p:text-sort` (pg. 143) - Sorts lines in a text document.
- `p:text-tail` (pg. 146) - Returns lines from the end of a text document.

A Error codes

All errors are in the `http://www.w3.org/ns/xproc-error` namespace (recommended prefix: `err`).

Error code	Description
`XC0001`	It is a dynamic error if the value of option `override-content-type` is not a text media type.
`XC0003`	It is a dynamic error if a "`username`" or a "`password`" key is present without specifying a value for the "`auth-method`" key, if the requested `auth-method` isn't supported, or the authentication challenge contains an authentication method that isn't supported.
`XC0007`	It is a dynamic error if any key in `parameters` is associated to a value which is not an instance of the XQuery 1.0 and XPath 2.0 Data Model, e.g. with a map, an array, or a function.
`XC0008`	It is a dynamic error if the stylesheet does not support a given mode.
`XC0009`	It is a dynamic error if the specified XQuery version is not available.
`XC0011`	It is a dynamic error if the specified schema version is not available.
`XC0012`	It is a dynamic error if the contents of the directory path are not available to the step due to access restrictions in the environment in which the pipeline is run.
`XC0013`	It is a dynamic error if the pattern matches a processing instruction and the new name has a non-null namespace.
`XC0014`	It is a dynamic error if the XML namespace (`http://www.w3.org/XML/1998/namespace`) or the XMLNS namespace (`http://www.w3.org/2000/xmlns/`) is the value of either the `from` option or the `to` option.
`XC0017`	It is a dynamic error if the absolute path does not identify a directory.
`XC0019`	It is a dynamic error if the documents are not equal according to the specified comparison `method`, and the value of the `fail-if-not-equal` option is `true`.
`XC0023`	It is a dynamic error if the selection pattern matches a wrong type of node.
`XC0024`	It is a dynamic error if the selection pattern matches a document node and the value of the `position` is "`before`" or "`after`".
`XC0025`	It is a dynamic error if the selection pattern matches anything other than an element or a document node and the value of the `position` option is "`first-child`" or "`last-child`".
`XC0029`	It is a dynamic error if an XInclude error occurs during processing.
`XC0030`	It is a dynamic error if the response body cannot be interpreted as requested (e.g. `application/json` to override `application/xml` content).
`XC0032`	It is a dynamic error if more than one document appears on the `source` port of the `<p:os-exec>` step.
`XC0033`	It is a dynamic error if the command cannot be run.
`XC0034`	It is a dynamic error if the current working directory cannot be changed to the value of the `cwd` option.
`XC0036`	It is a dynamic error if the requested hash algorithm is not one that the processor understands or if the value or parameters are not appropriate for that algorithm.
`XC0037`	It is a dynamic error if the `value` provided is not a properly `x-www-form-urlencoded` value.
`XC0038`	It is a dynamic error if the specified xslt version is not available.
`XC0039`	It is a dynamic error if the source port does not contain exactly one XML document or one HTML document if XSLT 1.0 is used.
`XC0050`	It is a dynamic error the file or directory cannot be copied to the specified location.
`XC0053`	It is a dynamic error if the `assert-valid` option on `<p:validate-with-nvdl>` is `true` and the input document is not valid.
`XC0054`	It is a dynamic error if the `assert-valid` option is `true` and any Schematron assertions fail.
`XC0055`	It is a dynamic error if the implementation does not support the specified mode.
`XC0056`	It is a dynamic error if the stylesheet does not provide a given template.
`XC0058`	It is a dynamic error if the `all` and `relative` options are both `true`.
`XC0059`	It is a dynamic error if the QName value in the `attribute-name` option uses the prefix "`xmlns`" or any other prefix that resolves to the namespace name `http://www.w3.org/2000/xmlns/`.
`XC0060`	It is a dynamic error if the processor does not support the specified `version` of the UUID algorithm.
`XC0062`	It is a dynamic error if the `match` option matches a namespace node.
`XC0063`	It is a dynamic error if the `path-separator` option is specified and is not exactly one character long.

Error code	Description
XC0064	It is a dynamic error if the exit code from the command is greater than the specified `failure-threshold` value.
XC0069	It is a dynamic error if the `properties` map contains a key equal to the string "`content-type`".
XC0071	It is a dynamic error if the `<p:cast-content-type>` step cannot perform the requested cast.
XC0072	It is a dynamic error if the `<c:data>` contains content is not a valid base64 string.
XC0073	It is a dynamic error if the `<c:data>` element does not have a `@content-type` attribute.
XC0074	It is a dynamic error if the `content-type` is supplied and is not the same as the `@content-type` specified on the `<c:data>` element.
XC0076	It is a dynamic error if the comparison `method` specified in `<p:compare>` is not supported by the implementation.
XC0077	It is a dynamic error if the media types of the documents supplied are incompatible with the comparison `method`.
XC0078	It is a dynamic error if the value associated with the "`fail-on-timeout`" is associated with `true()` and a HTTP status code `408` is encountered.
XC0079	It is a dynamic error if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.
XC0080	It is a dynamic error if the number of documents on the `archive` does not match the expected number of archive input documents for the given `format` and `command`.
XC0081	It is a dynamic error if the format of the archive does not match the format as specified in the `format` option.
XC0083	It is a dynamic error if the `@namespace` attribute is specified, the `@name` contains a colon, and the specified namespace is not the same as the in-scope namespace binding for the specified prefix.
XC0084	It is a dynamic error if two or more documents appear on the `p:archive` step's `source` port that have the same base URI or if any document that appears on the `source` port has no base URI.
XC0085	It is a dynamic error if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.
XC0086	It is a dynamic error for any unqualified attribute names to appear on a `<c:param-set>` element.
XC0087	It is a dynamic error if the `@namespace` attribute is not specified, the `@name` contains a colon, and the specified prefix is not in the in-scope namespace bindings.
XC0089	It is a dynamic error if the sequence type is not syntactically valid.
XC0090	It is a dynamic error if an implementation does not support directory listing for a specified scheme.
XC0092	It is a dynamic error if as a consequence of changing or removing the namespace of an attribute the attribute's name is not unique on the respective element.
XC0093	It is a dynamic error if a static error occurs during the static analysis of the XSLT stylesheet.
XC0094	It is a dynamic error if any document supplied on the source port is not an XML document, an HTML documents, or a Text document if XSLT 2.0 is used.
XC0095	It is a dynamic error if an error occurred during the transformation.
XC0096	It is a dynamic error if the transformation is terminated by XSLT message termination.
XC0098	It is a dynamic error if a dynamic XPath error occurred while applying sort-key to a line.
XC0099	It is a dynamic error if the result of applying `sort-key` to a given line results in a sequence with more than one item.
XC0100	It is a dynamic error if the document on port `manifest` does not conform to the given schema.
XC0101	It is a dynamic error if a document appearing on port `source` cannot be represented in the XDM version associated with the chosen XQuery version, e.g. when a JSON document contains a map and XDM 3.0 is used.
XC0102	It is a dynamic error if any key in option `parameters` is associated to a value that cannot be represented in the XDM version associated with the chosen XQuery version, e.g. with a map, an array, or a function when XDM 3.0 is used.
XC0103	It is a dynamic error if any error occurs during XQuery's static analysis phase.
XC0104	It is a dynamic error if any error occurs during XQuery's dynamic evaluation phase.

Error code	Description
XC0105	It is a dynamic error if an XSLT 1.0 stylesheet is invoked and option `parameters` contains a value that is not an atomic value or a node.
XC0106	It is a dynamic error if duplicate keys are encountered and option `duplicates` has value "`reject`".
XC0107	It is a dynamic error if a document of a not supported document type appears on port `source` of `p:json-merge`.
XC0108	It is a dynamic error if any prefix is not in-scope at the point where the occurs.
XC0109	It is a dynamic error if a namespace is to be removed from an attribute and the element already has an attribute with the resulting name. For instance, removing the namespace with the `ns1` prefix will raise this error when applied to `<something ns1:status="ok" status="bad"/>`.
XC0110	It is a dynamic error if the evaluation of the XPath expression in option `key` for a given item returns either a sequence, an array, a map, or a function.
XC0111	It is a dynamic error if a document of an unsupported document type appears on port `source` of `p:json-join`.
XC0112	It is a dynamic error if more than one document appears on the port `manifest`.
XC0113	It is a dynamic error if an attempt is made to delete a non-empty directory and the `recursive` option was set to `false`.
XC0114	It is a dynamic error if the directory referenced by the `href` option cannot be created.
XC0115	It is a dynamic error if the resource referenced by the `target` option is an existing file or other file system object.
XC0116	It is a dynamic error if the temporary file could not be created.
XC0117	It is a dynamic error if a report-format option was specified that the processor does not support.
XC0118	It is a dynamic error if an archive manifest is invalid according to the specification.
XC0119	It is a dynamic error if `flatten` is neither "`unbounded`", nor a string that may be cast to a non-negative integer.
XC0120	It is a dynamic error if the `relative-to` option is not present and the document on the `source` port does not have a base URI.
XC0121	It is a dynamic error if a document appearing on the `secondary` port has a base URI that is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .
XC0122	It is a dynamic error if the given method is not supported.
XC0123	It is a dynamic error if any key in the "`auth`" map is associated with a value that is not an instance of the required type.
XC0124	It is a dynamic error if any key in the "parameters" map is associated with a value that is not an instance of the required type.
XC0125	It is a dynamic error if the key "`accept-multipart`" as the value `false()` and a multipart response is detected.
XC0126	It is a dynamic error if the XPath expression in `assert` evaluates to `false`.
XC0127	It is a dynamic error if the `headers` map contains two keys that are the same when compared in a case-insensitive manner.
XC0128	It is a dynamic error if the URI's scheme is unknown or not supported.
XC0129	It is a dynamic error if the requested HTTP version is not supported.
XC0131	It is a dynamic error if the processor cannot support the requested encoding.
XC0132	It is a dynamic error if the override content encoding cannot be supported.
XC0133	It is a dynamic error if more than one document appears on the `source` port and a `content-type` header is present and the content type specified is not a multipart content type.
XC0134	It is a dynamic error if an implementation does not support `<p:file-info>` for a specified scheme.
XC0135	It is a dynamic error if `<p:file-info>` is not available to the step due to access restrictions in the environment in which the pipeline is run.
XC0136	It is a dynamic error if an implementation does not support `<p:file-touch>` for a specified scheme.
XC0137	It is a dynamic error if `<p:file-touch>` cannot be completed due to access restrictions in the environment in which the pipeline is run.

Error code	Description
XC0138	It is a dynamic error if an implementation does not support <p:file-create-tempfile> for a specified scheme.
XC0139	It is a dynamic error if <p:file-create-tempfile> cannot be completed due to access restrictions in the environment in which the pipeline is run.
XC0140	It is a dynamic error if an implementation does not support <p:file-mkdir> for a specified scheme.
XC0141	It is a dynamic error if <p:file-mkdir> not available to the step due to access restrictions in the environment in which the pipeline is run.
XC0142	It is a dynamic error if an implementation does not support <p:file-delete> for a specified scheme.
XC0143	It is a dynamic error if <p:file-delete> is not available to the step due to access restrictions in the environment in which the pipeline is run.
XC0144	It is a dynamic error if an implementation does not support <p:file-copy> for a specified scheme.
XC0145	It is a dynamic error if <p:file-copy> is not available to the step due to access restrictions in the environment in which the pipeline is run.
XC0146	It is a dynamic error if the specified value for the override-content-types option is not an array of arrays, where the inner arrays have exactly two members of type xs:string.
XC0147	It is a dynamic error if the specified value is not a valid XPath regular expression.
XC0148	It is a dynamic error if an implementation does not support <p:file-move> for a specified scheme.
XC0149	It is a dynamic error if <p:file-move> is not available to the step due to access restrictions in the environment in which the pipeline is run.
XC0150	It is a dynamic error if evaluating the XPath expression in option test on a context document results in an error.
XC0151	It is a dynamic error if the document supplied on schema port is not a valid Schematron document.
XC0152	It is a dynamic error if the document supplied on schema port is not a valid XML schema document.
XC0153	It is a dynamic error if the document supplied on schema port cannot be interpreted as an RELAX NG Grammar.
XC0154	It is a dynamic error if the document supplied on nvdl port is not a valid NVDL document.
XC0155	It is a dynamic error if the assert-valid option on <p:validate-with-relax-ng> is true and the input document is not valid.
XC0156	It is a dynamic error if the assert-valid option on <p:validate-with-xml-schema> is true and the input document is not valid.
XC0157	It is a dynamic error if the href option names a directory, but the target option names a file.
XC0158	It is a dynamic error if the href option names a directory, but the target option names a file.
XC0159	It is a dynamic error if any key in the "auth" map is associated with a value that is not an instance of the required type.
XC0160	It is a dynamic error if any key in the "parameters" map is associated with a value that is not an instance of the required type.
XC0161	It is a dynamic error if the first document on the source port does not conform to the required format.
XC0162	It is a dynamic error if the email cannot be send.
XC0163	It is a dynamic error if the selected version is not supported.
XC0164	It is a dynamic error if the document supplied on schema port is not a valid JSON schema document in the selected version.
XC0165	It is a dynamic error if the assert-valid option on <p:validate-with-json-schema> is true and the input document is not valid.
XC0166	It is a dynamic error if the requested document cannot be produced.
XC0167	It is a dynamic error if the requested document cannot be produced.
XC0200	It is a dynamic error if the pipeline input to the p:run step is not a valid pipeline.
XC0201	It is a dynamic error if the <p:uncompress> step cannot perform the requested content-type cast.
XC0202	It is a dynamic error if the compression format cannot be understood, determined and/or processed.

Error code	Description
XC0203	It is a dynamic error if the specified boundary is not valid (for example, if it begins with two hyphens "--").
XC0204	It is a dynamic error if the requested content-type is not supported.
XC0205	It is a dynamic error if the source document cannot be parsed by the provided grammar.
XC0206	It is a dynamic error if the dynamically executed pipeline implicitly or explicitly declares a primary input port with a different name than implicitly or explicitly specified in the p:run invocation.
XC0207	It is a dynamic error if the dynamically executed pipeline implicitly or explicitly declares a primary output port with a different name than implicitly or explicitly specified in the p:run invocation.
XC0210	It is a dynamic error if the `assert-valid` option on `<p:validate-with-dtd>` is `true` and the input document is not valid.
XC0211	It is a dynamic error if more than one document appears on the `grammar` port.
XC0212	It is a dynamic error if the grammar provided is not a valid Invisible XML grammar.
XD0011	It is a dynamic error if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.
XD0023	It is a dynamic error if a DTD validation is performed and either the document is not valid or no DTD is found.
XD0036	It is a dynamic error if the supplied value of a variable or option cannot be converted to the required type.
XD0043	It is a dynamic error if the `dtd-validate` parameter is `true` and the processor does not support DTD validation.
XD0049	It is a dynamic error if the text value is not a well-formed XML document
XD0057	It is a dynamic error if the text document does not conform to the JSON grammar, unless the parameter liberal is true and the processor chooses to accept the deviation.
XD0058	It is a dynamic error if the parameter duplicates is reject and the text document contains a JSON object with duplicate keys.
XD0059	It is a dynamic error if the parameter map contains an entry whose key is defined in the specification of `fn:parse-json` and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter `escape` with `true()` is also present.
XD0060	It is a dynamic error if the text document can not be converted into the XPath data model
XD0062	It is a dynamic error if the `@content-type` is specified and the document-properties has a "`content-type`" that is not the same.
XD0064	It is a dynamic error if the base URI is not both absolute and valid according to RFC 3986 (https://www.rfc-editor.org/info/rfc3986) .
XD0070	It is a dynamic error if a value is assigned to the `serialization` document property that cannot be converted into `map(xs:QName, item()*)` according to the rules in section "QName handling" of XProc 3.0 (https://xproc.org/) .
XD0078	It is a dynamic error if the loaded document cannot be represented as an HTML document in the XPath data model.
XD0079	It is a dynamic error if a supplied content-type is not a valid media type of the form " *type/subtype+ext* " or " *type/subtype* ".

B Namespaces used

- `http://www.w3.org/ns/xproc` (recommended prefix: `p`)
  The main XProc namespace, used for all of its elements, steps and some of its attributes.
- `http://www.w3.org/ns/xproc-step` (recommended prefix: `c`)
  This namespace is used for documents that are inputs or outputs from several standard and optional steps.
- `http://www.w3.org/ns/xproc-error` (recommended prefix: `err`)
  This namespace is used for XProc errors.
- `http://www.w3.org/2001/XMLSchema` (recommended prefix: `xs`)
  This namespace is used for data types, such as `xs:string` and `xs:boolean`.
- `http://www.w3.org/ns/xproc-http` (recommended prefix: `rh`)
  The namespace used for specifying request headers in document-properties in the `p:http-request` step.